Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Informatica
Programming Technology Lab

Modularizing Language Constructs: A Reflective Approach

Ph.D. Dissertation

Thomas Cleenewerck

Promotors: Prof. Dr. Theo D'Hondt and Dr. Pascal Costanza

July 3rd 2007

Abstract

Programming languages are in a continuous state of flux in order to keep up with emerging needs of programmers. They are grown with new constructs so that programmers can express the problems from their domain within the language they are using. Growing languages means to grow their implementations along with them. To support this, we wish to preserve the decomposition of languages into language constructs in their implementations. As the design of a language implementation directly reflects our intuitive decomposition, a developer can engage in the natural process of developing a language.

We preserve the decomposition into language constructs by modularizing the definition of language constructs in separate implementation modules containing their syntactical representation and their translational semantics. In this setting, growing a language boils down to writing or selecting the appropriate language constructs and establishing the necessary interactions. As the language is continuously evolving during its implementation and future evolutions, the modularization of the language constructs renders the implementation less susceptible to the continuous changes.

The modularization of language implementations has been the subject of much research in the domain of compiler technology. The complexity of this research lies in the fact that language constructs intrinsically take into account other language constructs and therefore compromise their opportunities for modularization. Indeed, the mechanisms presented by the contemporary state of the art technologies for separating a language implementation into modules do not suffice.

In this dissertation, we define a lightweight formal model for the modularization of language constructs. From this model we deduce a new language implementation design in which languages consists of three kinds of concerns: the *basic language concerns* defining the language constructs, the *language specification* defining the interactions between the basic concerns by using the *special-purpose concerns* which define the mechanisms to implement the interactions.

As a solution for the above model, we present an open design for a new lan-

guage development technique: A language implementation is decomposed into a set of interacting language modules called linglets. Each linglet defines in isolation the syntax and the (translational) semantics of a single language construct in terms of another (lower level) language. The mechanisms to establish the necessary interactions among the language constructs are captured in interaction strategies. Interaction strategies are defined as extensions of a specifically tailored metaobject protocol (MOP).

Linglets and interaction strategies can be reused across language implementations. In addition, new linglets and interaction strategies can be defined, and existing ones can be specialized to respectively establish the interactions with other linglets and to meet and adapt the interaction strategy for the challenges in a particular language implementation.

We validate our approach by developing a non-trivial family of domain-specific languages using a shared pool of language constructs and interaction strategies, and by implementing the necessary metalanguages.

This dissertation enables us to optimize language implementations with respect to separation of concerns according to their language constructs.

Nederlandstalig Abstract

Programmeertalen zijn continu onderhevig aan veranderingen gestuurd door de nieuwe behoeften van programmeurs. Hierdoor groeien ze omdat ze worden uitgebreid met nieuwe taalconstructies zodat programmeurs hun problemen van een bepaald domein beter kunnen uitdrukken. Als talen groeien, dan moeten uiteraard hun implementaties meegroeien. Om dit te ondersteunen willen we de implementaties van talen structureren op dezelfde wijze als talen gestructureerd zijn, namelijk volgens hun taalconstructies. Taalimplementaties die op die manier gestructureerd zijn, kunnen makkelijker en op een natuurlijke manier ontwikkeld worden.

Taalimplementaties worden volgens taalconstructies gestructureerd door elke taalconstructie te modulariseren in aparte taalmodules. Iedere taalmodule beschrijft de syntax en de translationele semantiek (uitgedrukt in termen van een andere taal) van één taalconstructie. Dit laat toe om talen en hun implementaties samen te laten groeien door nieuwe taalmodules te ontwikkelen en/of bestaande te selecteren en hun onderlinge interacties vast te leggen. Bovendien maakt deze modularisatie taalimplementaties minder kwetsbaar voor de continue veranderingen.

Modularisatie van taalimplementaties werd grondig onderzocht in het onderzoeksveld van compilertechnologie. De complexiteit van dit onderzoek ligt in de intrinsieke verbanden tussen taalconstructies die modularisatie bemoeilijken. We stellen echter vast dat de mechanismen aangeboden door de hedendaagse spitstechnologieën voor taalontwikkeling niet volstaan om taalimplementaties te structureren volgens de taalconstructies.

In deze verhandeling, definiëren we een formeel model van de modularisatie van taalconstructies. Op basis van dit model leiden we een nieuw ontwerp af dat een leidraad is voor de implementatie van talen en taalonwikkelingstechnologie. Het ontwerp schrijft voor dat talen opgebouwd worden uit drie soorten bekommernissen: de basis bekommernissen die de taalconstructies definiëren, de taalspecificatie die de interacties tussen de taalconstructies realiseren met behulp

van speciale bekommernissen die op hun beurt de mechanismen definiëren om de interacties te realiseren.

Als oplossing voor het formele model en het nieuwe ontwerp, stellen we een open nieuwe taalontwikkelingstechnologie voor waarin talen geïmplementeerd worden als een verzameling interagerende taalmodules die linglets genoemd worden. Elke linglet definieert, in totale afzondering van de rest van de taal en de andere taalconstructies, de syntax en de translationele semantiek van één enkele taalconstructie. De mechanismen om de interacties tussen de taalconstructies te realiseren zijn gedefinieerd in interactiestrategieën. Deze worden gemodelleerd als uitbreidingen van een specifiek gecreëerd metaobject protocol voor taalontwikkelingstechnologieën.

Linglets en interactiestrategieën kunnen hergebruikt worden in verschillende taalimplementaties. Daarenboven kunnen nieuwe gedefinieerd worden en kunnen bestaande gespecialiseerd worden zodat linglets kunnen interageren met andere linglets en zodat interactiestrategieën aangepast kunnen worden aan de uitdagingen die een concrete taalimplementatie stelt.

De nieuwe aanpak om talen te implementeren die we voorstellen in dit werk valideren we door een niet-triviale familie van domein-specifieke talen te implementeren gebruikmakende van een gedeelde verzameling van taalconstructies en interactiestrategieën.

Met onze aanpak is het nu mogelijk om de opdeling van taalimplementaties in taalmodules, die elk een taalconstructie definiëren, te optimaliseren.

Acknowledgments

The most thrilling aspect of doing research is to venture and to explore. The Programming Technology Lab (PROG) has given me the freedom and the means to do so. For this I want to thank prof. dr. Theo D'Hondt. I especially want to thank him for giving me the opportunity to investigate my ideas, despite their crudeness and sketchiness at times. Although this dissertation is longer than others, Theo took the necessary time to review the text. Theo, I still owe you a pencil.

My co-promotor dr. Pascal Costanza and myself crossed paths when I was consolidating my work. It is no exaggeration to say that Pascal played a key role in that process. In our technical discussions, he maintained the overview in the midst of a pool of very diverse features and ideas and he was able to quickly identify and grasp the interesting parts of my work. He is also the guy that worked his way through elaborate drafts of my text. Thanks to him, I learned the meaning of the sentence "less is better". Working with Pascal is pleasant as he likes technical details as well as the big picture of what is going on, but most importantly he quickly knows how and what to contribute to a working relation and to a person. Many thanks, Pascal and I hope we can continue our cooperation in the future.

I thank Görel Hedin, Tom Tourwé, Wim Vanderperren and Wolfgang De Meuter for finding the time to be on my thesis committee and for their insightful comments and suggestions.

During my activities at PROG I closely worked with Johan Brichau, Dirk Deridder, Johan Fabry and Elisa Gonzalez Boix. Johan Brichau helped me to shape some of my initial ideas. I recall his rigor, and persistence and our vivid discussions. Johan, thank you for this and for the proofreading at the end of the ride. You certainly returned the favor. Dirk Deridder, thank you for saving the day by helping me out when I was in need for a proofreader, and by offering a listening ear. Johan Fabry is a swift, no-fuss and to the point worker. His direct but fair style makes working with him rewarding and fun. Most importantly,

Johan has always been a great supporter. Thank you for that. Elisa Gonzalez Boix, both professionally and personally, you made a difference. I hope I can return the many favors with the same enthusiasm and kindness. Many thanks to the other members of my lab which also proof-read the thesis: Brecht Desmet, Charlotte Herzeel, Coen De Roover, Ellen Van Paesschen, Jessie Dedecker, Kris Gybels and Sofie Goderis. A thank you is also more than appropriate to all my colleagues at PROG who provided me with the opportunity to focus entirely on writing my dissertation.

Special thanks goes to my wife Veerle. She believed in me from the very start on, and coped with me in all these years when I was wandering off figurative and literally pursuing ideas. Last but not least, I wish to thank my parents and parents in-law for unconditionally supporting me.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
ACTA	formal model for ATMS
AGG	Algebraic Graph Rewriting
AMDD	Agile Model-driven Development
AOP	Aspect-oriented programming
AP	Adaptive Programming
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
ATMS	Advanced Transaction Models
BNF	Backus-Naur Form
DCO	Delegating Compiler Object
DSD	Domain-specific Description
DSL	Domain-specific Language
EBNF	Extended Backus-Naur Form
ELAN	A rewrite rule system
EUP	Enterprise Unified Process
FORTRAN	FORmula TRANslation
GPL	General Purpose Language
HTML	Hyper Text Markup Language
ICG	Integrative Composable Generators
ID	Identifier
IDE	Integrated Development Environment
INR	Incremental Non-local Results
IP	Intentional programming
IS	Implementation Strategies
ISC	Implementation Strategy Control
ISI	Implementation Strategy Interface
ISIM	Implementation Strategy Implementation Model

JAMOOS	A domain-specific language for language processing
JIT	Just In Time
JTS	Jakarta Tool Suite
KALA	Kernel Aspect Language for ATMS
LC	Language Construct
LDT	Language Development Technique
LHS	Left Hand Side
LISP	List Processing
LL	Linglet Language
LMOP	Linglet Metaobject Protocol
LMP	Logic Meta-Programming
LS	Language Specification
LSL	Language Specification Language
LTS	Linglet Transformation System
MDA	Model-driven Architecture
MDD	Model-driven Development
MOP	Metaobject Protocol
MPS	Meta-programming systems
NAT	Natural Number
OI	Open Implementations
OID	Open Implementation Design
OMG	Object Management Group
OO	Object Orientation
QVT	Query/View/Transformation
RCS	Relatively Consistent Schedules
RHS	Right Hand Side
RUP	Rational Unified Process
SDF	Syntax Definition Formalism
SIS	Semantic Implementation System
SOC	Separation of Concerns
SOP	Subject-oriented Programming
SQL	Structured Query Language
SSQ	Structure-shy Query
STS	Software Transformation Systems
T2SQL	Tuple Calculus to SQL
TAMPR	Transformation Assisted Multiple Program Realisation System
TXL	Tree rewriting Language
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

1.1 Research Context

We continuously use languages for expressing our thoughts. The Sapir-Whorf hypothesis [Who56, KK83] states that there is a systematic relationship between languages and our thoughts. The relationship is complex and a matter of much debate as it touches the essence of being human. Thinking and more precisely the use of symbolic representation is, what is generally believed, what separates us from the animals [Dea97]. One of the questions regarding language and thought which is important for this dissertation is, as Alan Ford has phrased the question [FP88], "Do we speak (have language) because we think, or do we think because we speak?".

It is like the chicken and egg question. Do we learn to think before we speak, or does language shape our thoughts? Experiments [Gor04, Spe04] have shown that both statements are likely to be true. In our early stages of development we learn the distinctions present in the language given to us. Shortly after we are born, after a couple of years we quickly become insensitive to what is irrelevant. In other words we learn what to ignore.

An example of how languages shape our thoughts is the following categorization experiment between Koreans and Americans [Gor04]. When Koreans and Americans see the same everyday events e.g. an apple in a bowl, a cap on a pen, they categorize them in accord with the distinctions of their languages. An American distinguishes between "in" and "on", while a Korean distinguishes between "tightly" or "loosely". Another example, suggesting that language shapes human thought is a counting study [Spe04] conducted with a Brazilian tribe whose language does not define numbers beyond two. Hunter-gatherers from the Pirahã tribe, were unable to reliably tell the difference between four objects placed in a row and five in the same configuration.

Other scientists in the field raise some interesting critiques about the counting study. Feigenson [Bie04] points out that there could be other additional reasons

why the Pirahã could not distinguish accurately between higher numbers, such as not being used to deal with large numbers or such tasks. This critique suggests that some external force to adapt is necessary. Indeed, through the ages it became clear that languages have been growing with new words, new rules and new semantics.

1.1.1 Programming Languages

The relationship between language and thought also affects the relationship between the programming languages and the programs we try to express in them: programming languages shape our programs, and programming languages themselves get shaped by programmers. Note that from this point on in the dissertation the term language refers to the term programming language unless explicitly stated otherwise.

Programming languages shape our thoughts, often without us realizing it. Consider for example a discussion [Jef] during the "Alternative Paradigms" session at the "Simple Design and Testing conference" [BDKT03]. Some people argued that Haskell and similar languages offer a way of programming that better reflects what developers think. However, this statement is contested with the argument that recursion is not part of how we naturally think: our brains are really bad at pushing things down and having them still in shape when we pop them back up. Although recursion is embedded in natural language, in programming languages recursion is a mathematical concept which needs to be taught to people.

Back in the 60thies Dijkstra reports in [Dij99] that 10% of his students had the greatest difficulty in coping with the concept of recursive procedures. As it turned out, those students had been priorly exposed to FORTRAN. The source of their difficulties was their operational view of programming in FORTRAN. Those students did not see how to implement recursion, so they could not understand it. Dijkstra even goes a step further and doesn't blame FORTRAN for not permitting recursion, but rather blames the fact that they had not been taught to distinguish the concept of recursion.

Languages get shaped as well. Many languages were designed and altered to meet the needs of their users. Before the invention of the modern computer, languages/formalisms were designed for providing mathematical abstractions for expressing algorithms, such as lambda calculus and Turing machines. With the advent of the first computer, the first languages used were different as they directly reflected operations and concepts of the physical machine in order to operate the machines. The second generation programming languages such as assembler, quickly abstracted from the machine instructions so that code can be fairly easily read and written by a human. On top of these languages, the very first so called general-purpose languages such as Fortran, Cobol and Lisp were constructed. There are many motivations for this move, but one of the underlying forces was

the gained experience. As the experience grew, standard concepts and solutions emerged and the desire and need increased for moving to a new language in which these concepts could be used directly. Although it took a while, the translation details became irrelevant for everyday programmers. These early general-purpose languages are still used today. Lisp[1], Fortran and Cobol underwent changes to this very day to reflect the needs of their programmers.

Clearly, programming languages continuously interact with the programs that developers try to express, and vice versa.

1.1.2 Growing Languages

The continuous interaction between programming languages and the programs that developers try to express, causes an ongoing effort in language design. This can be recognized in the tendency to raise the abstraction level of programming languages by increasing the expressiveness of the language with new language constructs.

Expressiveness of languages is the degree in which their language features and constructs can effectively convey the intentions of the programmer. The smaller the semantic gap between language concepts and the intentions of the programmer, the easier it gets to produce software. To this end, languages have been made more expressive. There are several informal ways to interpret the term expressiveness. Therefore we will briefly introduce the term.

Language constructs that merely introduce syntactic variations of the same intention are called syntactic sugar [Lan66]. In this dissertation we adopt the formalization of Felleisen [Fel91]. A language construct F is said to be more expressive compared to the language constructs $G_1, ..., G_n$ of a language L, if there does not exist a recursive homomorphic mapping[2] of $G_1, ..., G_n$ from $L \cup \{F\}$ to L. A homomorphic mapping from $L \cup \{F\}$ to L enforces that the translations of the existing language constructs G_i are structure preserving. This means that the existing constructs are mapped to constructs with exactly the same structure and the same semantics. Hence, if there does not exists a homomorphic mapping from $L \cup \{F\}$ to L, the translation of F has an *effect* on the semantics of the already existing language constructs. So, programs written in a more expressive language have a different global structure from the functionally equivalent programs (i.e. the programs obtained by applying the non-homomorphic mapping) in a less expressive language.

The conciseness conjecture [Fel91] states that programs in a more expressive programming language that use the additional facilities in a sensible[3] manner

[1]Lisp has been changed. An example of this are its scoping rules. However, overall Lisp has been mainly changed from within the language.

[2]recursive homomorphic mapping ϕ from $L \cup \{F\}$ to L-phrases where $\phi(G_i(a_1, ..., a_n)) = G_i(\phi(a_1), ...\phi(a_n))$, with $1 \leq i \leq m$

[3]Sensible in this conjecture informally excludes the use of more expressive constructs for

contain fewer programming patterns than equivalent programs in less expressive languages. Programming patterns are code fragments needed to express an intention. The problem with patterns is that they are an obstacle to understand the intention of the program. Moreover, as programs in an more expressive language have different global structure, the intent of the program written in a less expressive language is much harder to unravel. The use of a more expressive language facilitates the development process and reduces the need for patterns by making programs more concise and abstract [Fel91]. According to the conciseness conjecture, an increase in the expressiveness of a language results in an increased level of abstraction.

Language growth is not only visible in mainstream contemporary general purpose languages such as Java [Ham], but also in languages operating in niche markets. An example of such a language is Lua which has a strong emphasis on embedded systems and games. Lua is a rapidly evolving industrial language. It was first released in 1993 and ever since every year a new version of the language has been released. Lua has which evolved from domain-speicific languages such as DEL and SOL and has incorportated many features of other languages in order to keep with the requirements of their developers [IdFC07].

With the advent of software development techniques such as domain-specific language engineering [vDKV00] and more recently model-driven development [TB03], increasing the level of abstraction of languages has become part of the contemporary software development process in general [Cam97, Amb02]. Domain-specific programming languages (or models in the case of model-driven development) are designed with appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [vDKV00]. Moreover, the domains of these languages are so called E-type systems [Leh96]. These systems are strongly connected to the real-world, and the problem that is being described by those languages simply cannot be 'completely' specified. In addition, the languages are, by definition, in a state of constant change to reflect the changes in the real world. An interesting example of such a language is Lua. Although it is a general purpose language, conferences like the Game Develpers Conference (GDC) dedicate entire sessions to discuss and investigate how Lua needs to evolve for next generation hardware and sofware architectures [Boi06].

1.2 Growing a Compiler

Growing languages means to grow their implementations along with them. It is argued that languages should be designed with a plan for growth [Ste99] in accordance with the language constructs in such a way that a minimum effort is necessary to evolve the language. *It is thus pertinent to provide sufficient means*

non-observable behavior.

for controlling and supporting that process. This statement captures precisely the intent of this dissertation.

As languages are grown with new constructs so must their implementations be grown. By preserving the decomposition of a language implementation in terms of its language constructs, a language and its implementation directly reflect our intuitive decomposition. In addition, a language implementation can be grown similar to the way the language is grown. More precisely, a language implementation can be grown by incrementally adding and changing implementation modules, each describing a language construct.

The advantages of preserving the decomposition of language constructs in language implementations are legion.

- *Understandability*: The structure of a compiler mimics our mental picture of a language, allowing language developers to quickly identify which module implements a given language construct. This increases understanding and facilitates reasoning about language implementations as language developers can then focus on one construct at a time [Dij76]. Examples illustrating this are the language specifications which are used as a reference guide, language tutorials which are used to explain a language and language courses which are used to teach languages. All these documents discuss languages according to their various language constructs: for each construct, its syntax and its semantics is given.

- *Evolvability:* Languages and their implementations are easier to evolve and to maintain as they can co-evolve in synch in terms of the same concepts, i.e. language constructs.

- *Extendibility:* Extensions of language implementations boil down to the addition of a separate module defining the new language constructs. Examples illustrating this are the documents explaining extension of languages. These documents are structured in terms of the new language constructs and their semantics e.g. extension of Java 1.4 [Mic].

- *Reusability:* Language constructs are reusable in various languages as the implementation of the language constructs is described in separate modules which can be used to construct other languages. An example illustrating this is jMock [FP06], where the developers were confronted with a constant stream of requests for new constructs. Most of them were not generic enough to be useful for a generic audience, so for every application domain a set of constructs is selected and new ones are designed in order to construct an appropriate language with them.

Quite early on in the history of programming, macros facilities of extendible languages like e.g. C, Lisp [Gra94], ML [GST01] preserve the decomposition of

language constructs in language extensions. More so, some pioneering language development techniques like intentional programming [Sim95a, Sim96] and delegating compiler objects [BD, Bos97] that took initial steps towards preserving the decomposition of language constructs in language implementations, have also argued the benefits of such designs:

- *Understandability*: In extensible languages, the implementations of language extensions mimic our mental picture of language extensions as the various macros each correspond to a new construct. Language designers can thus focus on one construct at a time.
- *Evolvability:* Extendible languages are successful because a language extension and its implementation are grown together as each macro both defines a syntactical construct and the semantics of that new construct.
- *Extendibility:* Macro facilities showed that languages can be easily extended [CF04] as each macro extends the language with a syntactical construct and extends the language implementation with the semantics of that new construct.
- *Reusability:* Macros developed for one application context are reused in other contexts in conjunction with other macros.

In this dissertation we focus on the question how to grow a compiler just like languages are grown i.e. how to define a compiler by using modules where each module implements a language construct. The work presented here is a platform and a model to discuss and assess the improved understandability and reasoning, the ease of extendibility and evolvability, the increased reusability, and even to take a first step towards an iterative development process steered by successively adding language constructs. However, a detailed study of these benefits is beyond the scope of this dissertation.

Note that the decomposition of language implementations in terms of language constructs does not necessarily exclude other decompositions for example according to a functional phase like naming analysis. Such decompositions can be modeled via a language extension mechanism on top of the decomposition into language constructs similar to the way semantic aspects are defined in attribute grammars [HM03, Swi, OdMS00, Paa95]. More details can be found when in one of our directions of future work in Section 8.3.10.

1.3 Problem Statement: Modularization According to Language Constructs

In the previous sections we argued that in order to grow languages and their implementations in terms of language constructs, we need to preserve the de-

composition into language constructs. We can preserve that decomposition by modularizing the definition of language constructs in separate implementation modules containing their syntactical representation and their (translational) semantics of a single language construct in terms of another (lower level) language. Language constructs are defined as the syntactical constructs of a language having a distinguishable semantics with respect to the other language constructs of that language. In this dissertation, aware of the different granularity of modules defining language constructs in [Paa95, Bos97], a language construct can range from a single production to several non-terminals and the set of productions defining them.

A language can then be defined as a set of interacting language modules. In this setting, growing a language boils down to writing or selecting the appropriate language constructs and establishing the necessary interactions. As the language is continuously evolving during its implementation and future evolutions, the modularization of the language constructs renders the implementation less susceptible to the continuous changes.

1.3.1 Major Challenges

Modularizing a language implementation according to its language constructs is a challenging undertaking. The challenge is due to the fact that each language construct must be defined in isolation with respect to other language constructs from the language it will be used in. The three major challenges are:

M1 The first challenge lies in the grammar, where the syntax of the language constructs should be described in terms of other language constructs. The syntactical definition of the various language constructs must be modularized to involve only a single language construct.

M2 The second challenge lies in the definition of the translational semantics in terms of a target language. As described in Section 1.1.2, whenever the abstraction level is raised, the semantics of the language construct written in terms of the target language cannot be expressed with a homomorphic mapping. Therefore a more complex mapping must be used for expressing the translational semantics, which necessitates (by definition) the involvement of other language constructs. In other words, such language constructs have broadly scoped requirements. Consequently, in general, language constructs *intrinsically* take into account other language constructs and therefore endanger their opportunities for modularization.

M3 The third challenge is the *modularization of the mechanisms* for establishing the necessary interactions among the language constructs without breaking the modularization of the language construct.

1.3.2 Modularization Problems in Contemporary Language Development Techniques

The typical architecture of compilers [SWW+88] is subdivided in a set of functional layers such as lexing, parsing, analysis, byte code generation, optimization. Besides the architecture, common practices and techniques evolved in today's de facto standard such as the data structures for representing programs using abstract syntax trees, tree annotations, tree walks (i.e. traversals) to name a few. Language development techniques (LDTs) such as attribute grammars [Knu68, Paa95], Backus-Naur Form parser generators (such as Yacc [Joh79]) and rewrite rules [Klo92] evolved into formalized language development paradigms. These paradigms boosted the development of languages, as they free us from tedious tasks such as lexing, parsing, pattern matching, subsequent manipulation and the transformation/the interpretation of an abstract syntax tree (AST) representing a program undergoing a transformation.

Despite the significant progress in compiler technology, the impact of the proposed modularization is significant as *contemporary LDTs are not designed with the extensibility of language constructs in mind*. First of all, the dominant decomposition of a compiler into small modules is functional. The modules capture the various functional passes required to convert a program into a new program. As a result, the implementation of the language constructs is scattered in these modules.

In the dissertation, we consider the major LDTs such as: tree-based rewrite rules, graph rewrite rules, macros, template-based approaches, attribute grammars, compositional generators, and several ad-hoc approaches. Each of these techniques is extensively discussed in Chapter 3. Upon evaluating the three modularization challenges M1, M2 and M3 against the LDTs we find that:

- The grammar formalisms lack the expressive power necessary for decoupling the syntactical definitions of each language construct.

- The contemporary LDTs decompose language implementations into a set of implicitly cooperating modules, and it is not always clear how to relate these modules to language constructs.

- Language developers do not have the proper means for separating the interactions among the language constructs.

  In order to render the implementation of the more complex translational semantics robust to changes, LDTs provide specific implementation mechanisms called *interaction strategies*. Interaction strategies reduce direct and explicit communication with other modules and as such, they provide a means to design the language with a plan of growth in terms of changes

in the set of language constructs. Examples of such interaction strategies are among others: attribute propagation rules [OdMS00], forwarding [WdMBK02], structure shy queries [Whi02], traversals [VKV03] and symbol tables[4] [App98].

We observed though that LDTs only offer their own particular interaction strategies. These contemporary interaction strategies do not suffice because:

S1 They are not generally applicable. Interaction strategies are designed for reducing the coupling between the language constructs and their semantics caused by a particular communication pattern. If the pattern deviates from what can be captured in an interaction strategy, the interaction strategy is useless.

S2 There is room for improvement. Interaction strategies can be optimized and tailored for concrete coupling problems.

S3 There is room for new interaction strategies. There are certain categories of translational semantics which are not at all supported.

S4 There is no silver bullet interaction strategy. As interaction strategies may have conflicting tradeoffs, it is unlikely to find an overall general-purpose interaction strategy that combines all the merits of the existing interaction strategies and eliminates their drawbacks.

Interaction strategies in contemporary LDT are actually embedded within the techniques. This confines or even prohibits the introduction of new interaction strategies or changes to an existing one. Because of this and because of the shortcomings of existing interaction strategies, language developers do not have the proper means for separating the interactions among the language constructs. Moreover, we observed that tailored interaction strategies for specific language implementations hardly find their way into general purpose LDTs.

We did not found any evidence that contemporary technologies for separating a language implementation into modules support the strict modularization of a language implementation into language constructs because the modules do not always relate to language constructs and the interaction strategies cannot sufficiently reduce the implicit cooperations of modules.

1.4 Thesis Statement

In this dissertation, we present an open design of a new LDT through a metaobject protocol which is capable of modularizing languages according to their language

[4]Symbol tables (also called environments) are usually used in compilers for semantic analysis. They maintain mappings from identifiers to type and locations.

constructs.

The new technique separates the implementation of the various language constructs into discrete[5] modules. The modules are defined in complete isolation with respect to other language constructs, and meet the challenges M1 and M2. They are responsible for representing a program fragment of a larger program and define the operations on that program fragment capturing all the behavior of that language construct throughout the compilation.

To establish the necessary interactions required by the complex translational semantics, language designers rely on the open design for using and defining their own interaction strategies. Interaction strategies are defined orthogonally to the language constructs in a way that complex translational semantics can be expressed without violating the modularization of language concerns. As the language constructs, interaction strategies are described in a discrete module. As such, we meet the modularization challenge M3.

Interaction strategies can be reused across various language implementations and further specialized for accommodating any particularities of the language implementation at hand. The open design is based on a metaobject protocol that gives users access to (and control over) the run-time behavior of the LDT. Interaction strategies extend and specialize the metaobject protocol specifications.

With our approach it is now possible for preserving the decomposition of language constructs in the language compilers and keep the interactions and the interaction strategies for implementing the interactions modularized.

1.5 Approach of the Dissertation

We start the dissertation with an extensive study of existing LDTs. In the study we reveal the strengths and weaknesses of the modularization mechanisms of each approach, but more importantly we identify the successful interaction strategies used for preserving that modularization.

In order to modularize the language constructs and their semantics, the architecture of their compilers is defined as the product of discrete language modules each defining a single language construct. It does not suffice to separate a language implementation into several files; the language modules are designed in isolation from one another by completely parameterizing them. We present a formal model for the modularization of languages constructs by imposing a series of requirements (see Section 1.5.1).

In addition to the language modules, the language implementations use interaction strategies which glue together the language constructs. The challenges for modularizing interaction strategies are analyzed in detail. Each of the contemporary LDTs presented in the first part of the dissertation, are subsequently subjected to a thorough investigation for determining the extent to what they

[5] individually separate and recognizable different

adhere to the formal model and are capable of separating the interaction strategies. The results of this investigation are further analyzed for distilling the design challenges for a new LDT.

Based on our model and our design challenges we design and implement a new LDT. First, the core of the technique capable of implementing the language constructs and their translational semantics is discussed. We detail how modularized language constructs are implemented (see Section 1.5.2) and how the modules are composed for establishing a coherent and consistent cooperative behavior in the language (see Section 1.5.3). Second, the metafacilities (see Section 1.5.4) of the system for implementing interaction strategies are detailed. Interaction strategies ensure that the necessary interactions among the language modules for expressing more complex translational semantics do not break the modularity of the language.

1.5.1 Modularization Model

Our model describes five requirements for modularizing language constructs: one requirement on the syntax, three requirements on the translational semantics and a fifth requirement on the program representation.

From this model we deduce a new language implementation design in which languages consist of three kinds of concerns: *basic language concerns* defining modularized language constructs, *language specifications* defining the interactions between basic concerns by using *special-purpose concerns* which define the mechanisms for implementing the interactions.

The new language implementation design meets all the modularization challenges listed in Section 1.3.1: the basic language concerns meet the challenges to modularize the syntax (M1) and the translational semantics (M2) of the language constructs, and the special-purpose concerns meet the challenge to modularize the mechanisms that establish the interactions among the language constructs (M3) and in order to overcome the shortcomings S1 to S4 of the existing interaction strategies.

Basic Concern

Each basic concern comprises a *modular* language construct which is defined in *isolation* with respect to the rest of the language implementation. It is defined by a syntactical definition and its translational semantics.

The first requirement modularizes the syntactical definition of a language construct. As such, the syntactical definition of a language module no longer directly refers to other syntax definitions belonging to other language modules, rendering it composable with other language modules. For example, a grammar production of an `if` statement in BNF [BBG+60] should be defined in terms of

a condition, a true and false branch instead of directly referencing the grammar productions `expression` and `statements`.

The challenges for modularizing the translational semantics of language constructs are due to the more complex translational semantics which necessitate the involvement of other language modules. We define three requirements that modularize:

- requests for external information e.g. looking up the memory location of a variable which was defined when the variable was declared.
- production of scattered code fragments e.g. integrating a piece of advice code in a method
- compositionality problems e.g. in an optimized translation conditional expressions are mapped to conditional jumps, but these jumps must be converted to a value when conditional expressions are used in the right hand side of assignments.

In order to adhere to the three requirements we use another definition of the translational semantics in which we distinguish between two concepts: *definition* and *effect*. The requirements exile the part of the semantics of a concern which involves other concerns to the language specification. We call this part the *effect* of the translational semantics of a concern. The remainder of the translational semantics in the concern merely *defines* the translational semantics. More precisely, the definition of the translational semantics *only*:

- states what information is required, but does retrieve the information e.g. the translation of a variable reference declares the need for the memory location but does not define how to lookup this memory location.
- states which results are produced that need to be scattered throughout the target program, but does not integrate them e.g. the translation of an advice only produces the advice code but does not integrate it in the targeted methods
- assumes that the semantics of the parts are compositional, but does not resolve compositionality conflicts e.g. in an optimized translation conditional expressions are mapped to conditional jumps, but when conditional expressions are used in the right hand side of assignments, the semantics of assignments assumes a value and does not convert these jumps to a value.

The fifth requirement enforces that the translational semantics of a language construct can produce incomplete program fragments. As such, the translational semantics can restrict itself to produce its equivalent target program only e.g. the translational semantics of method declarations should not retrieve and integrate

the pieces of advice code that must become a part of its method body. The produced target language program fragments are completed by other language modules. To complete them, the fragments need to be changeable. The translational semantics of every language module can be subjected to changes caused by other language modules. Therefore, language modules must also enforce the consistency of its equivalent program fragments as the invasive integration of two program fragments may easily lead to unexpected and undesired structural and behavioral conflicts, if the consistency of both generators is not ensured [Bri05]. Consistency enforcement is not additional in the sense that another concern pollutes the language module. In fact, consistency is implicitly present in the translational semantics. The only additional requirement lies thus in its visibility. In other words, consistency has to be explicitly formulated.

These five requirements ensure that the basic concerns meet the challenges for modularizing the syntax (M1) and the translational semantics (M2) of the language constructs.

Language Specification

The Language Specification concern or plainly the language specification complements the definition of the semantics to effect the translational semantics i.e. execute the part of the translational semantics which involves other language constructs.

Special-purpose Concerns

The special-purpose concerns are the mechanisms which are used to effect the translational semantics i.e. execute the part of the translational semantics which involves other language constructs. There are three kinds of special-purpose concerns, one for each requirement of the translational semantics. In order to effect the more complex translational semantics one must execute the concern violating behavior: compute its context information, scatter code fragments and resolve compositionality against other language constructs respectively.

- Context information is effected by stating how that information should be computed or retrieved.
- Scattered code fragments are effected by stating which other language modules should handle these results or by stating how to handle results from other language modules.
- Compositionality is effected by stating a solution to cope with compositionality problems.

The special-purpose concerns are separated from the basic concerns in order not to compromise the modularity of the latter. In other words, special-purpose concerns may not change or require changes to be made in the basic concerns.

The special-purpose concerns meet the challenges for modularizing the mechanism that establish the interactions between the language constructs (M3) because they are defined in separate concerns. The separated interaction strategies offers the opportunity for choosing the optimal interaction strategy and to overcome the shortcomings S1 to S4 of the existing strategies .

1.5.2 Modularized Implementation of Language Constructs

For the modularization of the implementation of the language constructs into discrete language modules we adopt a compiler architecture that is orthogonal to the current process-oriented architectures provided by contemporary LDTs. Each language module encompasses the definitions and the functionality of the various functional layers of a single language construct. A language module encompasses natively parts from the two basic functional layers: the concrete syntax definition and the translational semantics. Other layers can be added to the language module when required such as checking or pretty printing to name a few.

The language modules meet the challenges for modularizing the syntax (M1) and the translational semantics (M2) of the language constructs because:

- The syntax is expressed with higher order productions i.e. productions that take other productions as a parameter instead of directly referencing other productions.
- The effect of the semantics of language constructs is dependent on and expressed by changing the overall language semantics defined by other language constructs. By omitting how to effect their semantics, we are able to formulate the semantics solely in terms of the language construct itself, thereby preserving their modularity. In order to express this restricted formulation of semantics, the language modules are equipped with appropriate mechanisms for only *defining* their translational semantics. In correspondence to our modularization model (see Section 1.5.1), this boils down to
  - a mechanism for stating what information is required e.g. the translation of a variable reference states the need for the memory location,
  - a mechanism for declaring that results are scattered e.g. the translation of an advice only states that the advice code must be scattered,
  - a mechanism to define the semantics compositionally e.g. in an optimized translation conditional expressions are mapped to conditional jumps, but when conditional expressions are used in the right hand side of assignments, the semantics of assignments uses the semantics of the of its right hand side and assumes it is a value.

This way their effect can be provided when a language using that language module is defined.

The additional requirement of consistency, which is imposed by the formal model, is enforced by a mechanism to support that changes inflicted by other language modules do not corrupt the translational semantics.

We developed a new LDT and implemented a prototype transformation system called the *Linglet Transformation System* (LTS). In LTS, *linglets* implement the modularized language modules. This part of LTS is called the core or the kernel transformation system.

LTS is based on prototype-based programming languages, which offer linguistic support for elegantly capturing the definition of a language construct and offer the ability for customizing the behavior of individual code fragments. The latter is necessary for effectively modularizing the effects language constructs have on other language constructs e.g. code fragments can ensure their consistency and can integrate themselves in the target program.

1.5.3 Language Specification

Each individual linglet defines a single language construct accompanied by its translational semantics. In order to construct a language, the linglets are composed together in a Language Specification (LS). In order to define the overall language semantics, a LS composes and customizes the language modules for establishing the necessary interactions among them. The language specification is the sole place where language modules become aware of the existence of other language modules and their composition. As language modules combine syntactic descriptions with translational semantics, a composition determines both the grammar of the language and the semantics of the overall language. In other words, the language is defined via a single specification. Hence, a language definition are specified in single and separate concern.

The higher order productions, which describe the syntactical definition of the language constructs, take other productions as parameters, aka *syntactical parameters*. By binding the syntactical parameters of a language module to other language modules in the language specification, they are syntactically composed with one another.

Although the syntactical definition also determines the semantical composition, this is not sufficient in the case of the more complex translational semantics. The translational semantics can only be partially specified in the language modules for preserving the modularity of modules. The omitted effect of the semantics on other language modules is specified in the language specification, as the language specification is the only place where the whole language is known. Language modules are extended with additional behavior that completes their translational semantics:

- the computation of context information e.g. looking up the memory location of a variable which was determined at its declaration site.
- the handling of scattered results e.g. integrating the advice code in the targeted methods
- the resolution of compositionality conflicts e.g. in an optimized translation where conditional expressions are mapped to conditional jumps, the conversion of these jumps to a value when conditional expressions are used in the right hand side of assignments.

Implementing that behavior is challenging as, by definition, that behavior solely involves other language modules. A plain implementation would quickly lead to a rigid and fragile language specification. For this reason, the implementation of that behavior is encoded via (tailored) language implementation *interaction strategies*. These strategies reduce accidental coupling, avoid tangling and scattering and consequently turn the language specification into a flexible artifact.

1.5.4 Language Implementation Interaction Strategies

In order to effect the translational semantics of the language modules, context information must be computed, scattered results must be propagated to other language constructs, and compositionality conflicts need to be resolved. Interaction strategies are implementation mechanisms that capture a common solution pattern for implementing these effects such that accidental coupling, tangling and scattering are reduced to a minimum. A well known example of such an interaction mechanism which is used for retrieving context information is traversals. Traversals allow us to retrieve information which involves the entire hierarchical structure of a program fragment (term) in a controlled fashion. They locally define the actions that need to be applied to each encountered subterm e.g. rewrite the term or collect information, and declaratively specify the properties of a traversal i.e. the order, when the recursion should be continued or broken off, and the direction. A traversal visits all the subterms, however only the subterms where an action has to be performed are specified, hereby reducing the accidental coupling with all the other subterms.

Interaction strategies meet the challenges for modularizing the mechanism that establish the interactions between the linglets (M3) because they are implemented as discrete language implementation extensions. By adding these extensions to an LDT, the translational semantics of language modules can be completed in a language specification by the interaction strategies defined by these extensions.

As each interaction strategy is captured in a discrete extension, interaction strategies themselves are defined as a separate concern. As such, interaction

strategies can be shared and reused across various language implementations. But more importantly new interaction strategies can be defined, and existing ones can be specialized in order to meet and tweak an interaction strategy for the separation of concern challenge for a particular language implementation. By taking into account the specific language implementation, the language designer can exploit specific characteristics or structural properties of the language. Although this renders the interaction strategies dependent on the language, the result is an implementation that is optimized in terms of separation of concerns. As the optimal interaction strategy can be defined, interaction strategies approach the silver bullet strategy (S4) by overcoming the limited applicability of the existing interaction strategies (S1), through their improvement (S2) and the definition of new interaction strategies (S3).

Interaction strategies are able to improve the implementation of languages because they capture a common communication pattern between language modules. Any communication between language modules adhering to that pattern can be expressed with that interaction strategy. Future changes to the language, made in accordance with the communication rules captured by that pattern, do not invalidate the effect of the translational semantics of the language modules implemented by that interaction strategy. Hence, interaction strategies define a plan for growth for the language. Changes that slightly disagree with the current interaction strategy also benefit from interaction strategies as interaction strategies can be easily customized to support the new communication pattern.

In essence, interaction strategies control the information flow during the execution of the compiler. For example, an interaction strategy that provides information computed by or residing in one language module, to another language module requesting that information, e.g. a request of information by a module representing a code fragment where the request information resides in an ancestor of that code fragment. As the execution of the compilation process is entirely driven by discrete language modules, interaction strategies require control over the execution of the language modules. For example, intercept requests of information and redirect those requests to a language module which contains the requested information, e.g. intercept a request of information by a module representing a code fragment and redirect to an ancestor of that code fragment. To this end, we apply the principle of open implementation design (OID) and design a reflective layer on top the language modules. The resulting metaobject protocol, combines the benefits of two worlds: the modularized language constructs together with the ability for extending the transformation system with interaction strategies e.g. the module requesting information can remain modular as it is oblivious to the strategy required to lookup the requested information and the actual module containing the requested information.

We do not have to include *new features* in the core system, nor *pollute* the language modules with additional responsibilities. Therefore the core remains a system with a simple semantics. That kind of unanticipated control ensures the

separation of concerns of the interaction strategies and the basic concerns, and ensures the necessary latitude for constructing appropriate interaction strategies. In addition, the metaobject protocol supports the reuse of existing discretely defined interaction strategies through specialization and customization.

1.6 Contributions

The following are the major contributions of this dissertation:

1.6.1 Survey of Contemporary Language Development Techniques

We present a detailed and extensive study of the various approaches, formalisms and techniques used for implementing languages. We include a wide range of techniques including ad-hoc, generative, embedded, and compilation-based approaches. More precisely, we distill and discuss the characteristics that impact the modularization of the transformation process into a set of transformation modules: fine-grained granularity of transformation modules, and transformation scopes. We introduce, together with their strengths and weaknesses, each mechanism that the individual LDTs offer for modularizing their implementations and handle the different scopes of transformations. We identify the successful mechanisms as interaction strategies.

1.6.2 Modularization of Language Constructs Model

We describe a new model for the design of compilers which is orthogonal to contemporary approaches. The model modularizes the compiler according to the language constructs of the language. We successfully meet the three challenges for modularizing the grammar, the translational semantics and the interactions between the language constructs which are listed in Section 1.3.1. These modules form the basic concerns for constructing a language. Each module describes the syntax and the translational semantics. The interactions and communications among these basic concerns to effect their translational semantics are defined by using special-purpose concerns. The special-purpose concerns are separated from the basic concerns in order not to comprise the modularity of the latter.

1.6.3 Kernel Transformation System

We present a design of a *new* LDT as a solution for the requirements described by our formal model. The LDT we present in this dissertation is called the Linglet Transformation System (LTS). LTS modularizes the various basic language

concerns into linglets. A linglet can be adapted to complete is translational semantics with interactions with other linglets. To this end, LTS is implemented in a prototype-based object-oriented style. As such, LTS offer the linguistic support for elegantly capturing the definition of a language construct and also the ability for customizing the behavior of individual code fragments. Hence, our transformation system meets the modularization of syntax and semantics of language constructs (M1 and M2) stated in Section 1.3.1.

1.6.4 Metafacilities for Defining Interaction Strategies

We introduce and argue the need for a reflective layer for transformation systems so as to allow language designers for constructing and using the interaction strategies that best fit. The reflective layer is realized through a well designed metaobject protocol. Via that protocol, interaction strategies can exercise control over the language modules without requiring any changes to be made to the language modules. As such, the modularity of the interaction strategies and the basic concerns is guaranteed. In addition, the metaobject protocol supports the reuse of existing discretely defined interaction strategies through specialization and customization. Hence, we have designed a transformation system which meets the modularization of interactions among language constructs (M3) stated in Section 1.3.1.

1.6.5 New Strategies

We conducted two experiments in which we presented the interaction strategies not as a monolithic entity but rather as a family of interaction strategy extensions; one experiment for retrieving context information and another one for declaring and specifying the scattering of code fragments. The former is a *variation* of an existing interaction strategy from another LDT, the latter is an *entirely new* interaction strategy. We also indicate how the interaction strategies of other LDTs can be implemented. Hence, we further illustrate the flexibility and extensibility of the metaobject protocol for constructing and refining interaction strategies. We have overcome the interaction strategy shortcomings (S1 to S4) listed in Section 1.3.2.

1.7 Outline

The dissertation is structured as follows:

Chapter 2 introduces some basic terminology of language implementations. We start by arguing that the modularization of language constructs is a complex undertaking because language constructs in general do not compose.

Furthermore, we sketch the typical architecture and classify the various implementation approaches. We then approach the translational semantics of a language to the perspective of a modularized model.

Chapter 3 provides a study of the various kinds of LDTs. We focus on the extent to what a language implementation can be modularized in these LDTs. We highlight the mechanisms of the LDTs to separate a language implementation into modules as well as its mechanisms to use and combine the modules into a fully operational language implementation.

Chapter 4 presents a formal model that separates the language constructs and their translational semantics by a series of requirements. We impose one requirement on the program representation, three requirements on the valuation function and one requirement on the syntax of a language construct. From this model we deduce a new language design consisting of three concerns: the basic concerns, the special-purpose concerns and the language specification concerns. The basic concerns capture a modularized language construct. The special-purpose concerns are used for gluing together the basic concerns in the language specification concern. The challenges for modularizing the special-purpose concerns are analyzed in detail. We present a summary of the investigation of the degree to what the LDTs adhere to our modularization model. The results of this investigation are further analyzed in order for distilling the design challenges for a new LDT.

Chapter 5 describes the core of a new LDT called the Linglet Transformation System (LTS) which is based on our modularization model. In LTS, a language is conceived as a set of interacting language modules called linglets. Each linglet defines a modularized language construct consisting of its syntax and its translational semantics. We detail how modularized language constructs are implemented and how the modules are composed for establishing a coherent and consistent cooperative behavior in the language. Furthermore, we show that LTS fulfills the five requirements from Chapter 4.

Chapter 6 describes the metafacilities on top of the core system for implementing interaction strategies. Interaction strategies are used for establishing the interactions among the language modules. Besides a detailed description of the metafacilities, we also present a design rationale by discussing the characteristics of open implementations. The metafacilities are implemented by means of a metaobject protocol (MOP). The MOP in LTS is called LMOP. We conduct two experiments in which we implement two interaction strategies: one strategy for retrieving context information and the other one for declaring and specifying the scattering of code fragments. We also indicate

how the interaction strategies of other LDTs can be implemented. As such, we demonstrate the potential of LTS.

Chapter 7 validates our approach by implementing a family of domain-specific languages for transaction management. We incrementally develop the family by starting with a small language and growing it into a number of different languages. We have designed a set of reusable linglets and interaction strategies based on the LMOP. The linglets define the modularized language constructs while adhering to the five requirements postulated in Chapter 4. Using the shared set of linglets and interaction strategies, we defined five different languages by merely recombining the linglets. The interaction strategies provide us the means for changing composition of the linglets without having to re-implement all the interactions between them.

Chapter 8 concludes the dissertation and indicates directions for future research.

Appendix A presents a thorough investigation of each of the contemporary LDTs presented in Chapter 3, in order to determine the extent to what the LDTs adhere to our formal model.

Appendix B presents a detailed analysis of the applicability of the interaction strategies found in contemporary LDTs.

Appendix C gives a formal overview of the aspect language KALA for advanced transaction management. KALA is used in our validation for implementing a family of domain-specific languages for transaction management.

Chapter 2

Language Implementations

Programming languages provide a set of language constructs for describing a solution for a problem e.g. an if-statement or an addition expression. Many languages have been designed and altered in order to meet emerging needs of programmers so they can express the problems of their domain within a language.

Our focus is on the increase of expressiveness of a programming language by adding new or replacing existing language constructs. The goal of this dissertation is preserving the modularization of language constructs in the implementation of these languages.

As we will explain in more detail, increasing the expressiveness of a programming language raises a challenging problem for modularizing its implementation along its language constructs because new expressive language constructs require changes to be made to the definition of other language constructs and they do not compose in general. In the next two chapters we investigate the mechanisms and the features which are offered by contemporary development techniques for modularizing a language implementation. In Chapter 3, each individual contemporary language development technique is presented with its strengths and weaknesses. This chapter serves as an introduction for that presentation.

We start by introducing basic terminology and sketch a typical implementation architecture of a language implementation. As languages have been implemented since the dawn of computer science and have been thoroughly researched, there are a lot approaches, formalisms and techniques for implementing languages. This dissertation focuses on the semantics of languages in compilation-based language implementations.

In order better to understand the kind of systems which we consider as the background of this dissertation, we classify the various implementation approaches for implementing languages. The challenges we are facing in this dissertation are the characteristics that impact the division of the transformation process into a set of transformation modules. We distill and discuss the fine-grained granularity of transformation modules and the scopes of transformations.

In the first section, we elaborate on the impact of extending a language with new language constructs for raising expressiveness. Section 2.2 introduces the terminology used in this dissertation and sketches a typical implementation architecture of a language implementation. The section concludes with a classification of various implementation approaches for implementing languages. Subsequently, Section 2.3 discusses the semantics of a language from our perspective of modularized language constructs. We conclude this chapter in Section 2.4.

2.1 Modularization of Languages into Language Constructs

Programming Languages are the primary tools developers use for constructing software. Languages are in fact a crucial tool as they stipulate the concepts which can be used for reformulating a problem into a solution/implementation. Languages are representational devices whose merits of should be judged according to how generally, naturally, and easily problems and applications can be represented and can be reasoned about. Exactly how the notions "generally, naturally, and easily" correspond or can be realized with language design notions such as simplicity, expressiveness, consistency and completeness is a matter of common sense and of much debate [Gab91, CDE+05]. For the purpose of this dissertation we will take a closer look at the key enabler "expressiveness".

In this section, we discuss the impact of increasing the expressiveness of a programming language on the modularization of its implementation along its language constructs.

2.1.1 Expressiveness

Expressiveness of a language is the degree to which its language features and constructs can effectively convey the intentions of a programmer. The smaller the semantic gap between the intentions of a programmer and the language constructs and features of a language, the easier it gets for producing software. To this end, languages have been made more expressive. There are several informal ways to interpret the term "expressiveness". Language constructs that merely introduce syntactic variations of the same intention are called syntactic sugar [Lan66]. For example, in Lisp or Scheme, the `let*` statement is sometimes dismissed as syntactic sugar for a *let*. Syntactic sugar is therefore not considered to be an expressive extension of a language. This idea was further refined by Steele and Susman who considered constructs that are expressible with a syntactical local structure and that are behavior-preserving not as expressive language constructs. In this dissertation we adopt the formalization by Felleisen [Fel91].

2.1.2 Expressiveness Formalized

Felleisen defined a language L as a set of phrases which are a subset of all abstract syntax trees (terms) constructed with a number of function symbols $F_1, ...F_n$ with varying arities $a_1, a_2, ...$ The function symbols are the syntactical constructs defined by the grammar of L. The sentences (programs) of L are a recursive combination of its phrases bounded by the rules stipulated in the grammar of the language L. Furthermore let L' be the language L stripped of a language construct G of L.

Expressiveness is defined by using the concept of a homomorphic function mapping. A homomorphism is a map from one algebraic structure to another of the same type that preserves all the relevant structures and their operations. The structure in a language are the tuples defined by its function symbols (a.k.a the nodes of the abstract syntax trees), the operations are its function symbols.

A mapping is established between the language L and the language L' without a construct G. The properties of a mapping used between these languages, teaches us about the semantics of the construct G in terms of the language without that construct. In the case of a homomorphic mapping between L and L', it teaches us that the language construct G is translated into another language construct.

2.1. DEFINITION. *The language construct G is said to be **expressive** if there does not exist a recursive homomorphic function mapping ϕ from L-phrases to L'-phrases.*

$$
\begin{aligned}
&\textit{where}\\
&\qquad \phi(F(a_1, ..., a_n)) = F(\phi(a_1), ...\phi(a_n)),\ \textit{for each } F \in L'.\\
&\qquad L' = L - G
\end{aligned}
$$

The homomorphic mapping ϕ enforces that the translations of expressive language constructs are structure preserving. Indeed, every language construct of L that is also in L', is mapped to the exact same language construct of L'. The language constructs of L', do not have any *effect* on the already existing language constructs. However, if there is no such a homomorphic mapping, the programs written in a more expressive language L have a different global structure from functionally equivalent programs in a less expressive language L'.

The conciseness conjecture states that programs in a more expressive programming language that use the additional constructs in a sensible manner contain fewer programming patterns than equivalent programs in less expressive languages. The term "sensible", in this conjecture, informally excludes the use of more expressive constructs for non-observable behavior. The problem with patterns is that they are an obstacle for understanding the intention of a program. Moreover, as programs in an more expressive language have different global structure, the intent of a program written in a less expressive language is harder to

unravel. The use of more expressive language constructs facilitates the programming process by making programs more concise and abstract [Fel91].

2.1.3 Modularization

Growing a language can have a significant impact on the semantics of existing language constructs, as the addition of a new expressive language construct requires changes to be made to the definition of existing language constructs. The impact of this on the modularization of a language implementation according to its language constructs is that language constructs in general do not compose.

Consider for starters the following simple example of the addition of variables to a pure expression language without variables. The semantics of a variable usage or assignment is a value which is retrieved from, or assigned in, an environment. The environment needs be treaded through the evaluation process: the semantics of every language construct in the expression language need to be adapted to take into account the environment. In general, language constructs do not compose, unless they are designed to compose or unless they are complemented with additional logic. The latter case has been proven by Mark P. Jones and Luc Duponcheel in [JD93] where they show that monads (definitions of language constructs) cannot be composed as such, but need additional auxiliary functions. Later on, the idea to combine monads via monad transformers appeared in Moggis work [Mog97]. Jones et.al. [JD93] illustrate their approach by showing that in monadic programming the environment threading can be separated in an auxiliary function or transformer. These functions combine the monads that evaluate the expression language constructs and the monads that evaluate the variable usage and assignments.

Similar examples might be encountered in many practical scenarios as well. The expressive language construct which Felleisen [Fel91] used to illustrate his formalization of expressive language constructs was a construct for a transaction counter. In an imperative language such as Scheme, one can bind the transaction manager to a procedure which increments the counter variable defined in the lexical scope of the procedure. In a functional language, the counting must be realized in a different way as there are no destructive assignments. The transaction manager must return the new counter and the result of the transaction. At every call site the paired result must be disassembled. So the addition of the transaction counter requires an invasive change, distributed over the whole program.

A similar situation occurs when exception handling is added to a functional language. A single function in a program must be able to decide whether that program will continue to execute or an error has to be thrown. For this, the whole program has to be rewritten in continuation passing style. There are other means to implement exception handling, but continuation passing style localizes the semantics of whether to execute an error or not.

In the remainder of this chapter we introduce the suite of contemporary language development techniques and their terminology.

2.2 Language Development Techniques

Designing and implementing a language is an intellectual challenge of considerable complexity. There are many ways for implementing languages and for structuring their implementation. There is no consensus on the names that are used for grouping them. In Section 2.2.1, we list the names and their connotations, and conclude with a new term "Language Development Technique" (LDT). To be able to discuss and introduce these techniques in a coherent fashion, let us briefly sketch the typical architecture of a language implementation and its parts. The typical language implementation architecture emerged from the construction of the early general-purpose languages. To this very day, this early architecture is the dominating architecture and is therefore omnipresent in literature and practice. In order better to understand the kind of systems which we consider as the background of this dissertation, we classify the various implementation approaches for implementing languages. The various LDTs are classified by the degree of support they offer for controlling and systematically implementing a language.

2.2.1 Unifying Terminology

One very apparent manifestation of the non-consensus of a general technique for language implementations is the number of names given to techniques, systems and formalisms. Some of these terms started out as very general terms but were narrowed down as research progressed. Other terms underwent the opposite evolution, as concepts of specific systems are getting better understood and researched, these concepts (along with the early systems) were generalized. As a result, terms are overloaded and a general name to denote a systematic approach for the implementation of languages is no longer available.

Existing Terminology

The following list explains and sketches the background of the most common encountered names. This list does not present an exhaustive overview of the various techniques, systems and formalisms, but merely consists of the names that are used for grouping them. A suitable classification of the various techniques, systems and formalisms for this dissertation is given in Section 2.2.3.

Compiler Tools ([vDKV00]) and Language Technology (e.g. [BLS98])
A compiler is a computer program (or set of programs) that translates text

written in a language (the source language) into another language (the target language). The original sequence is usually called the source code while the output is called object code. The most common reason for wanting to translate source code is to create a highly optimized executable program. The term "compiler" is primarily used for programs that translate source code from a high level language to a lower level language (e.g. assembly language or machine language). Compiler tools encompass a wide range of implementation artifacts necessary for constructing a compiler. However, the term is also used for code translations of contemporary high level languages to lower level languages and from domain-specific languages (DSLs) [vDKV00] to general purpose languages (GPLs) respectively.

Compiler-compilers [Paa95] Compiler-compilers, also called compiler writing systems or translator writing systems, are systems that produce fully functional language implementations based on higher level descriptions of a compiler. The first to use that name was Tony Brooker [Paa95] in 1960. The term was very popular in the 70ties with the advent of the first attribute grammar implementations by Fang in 1972. In their early years the systems produced compilers only, nowadays the term is also used for mere parser generators such as Yacc [Joh79] (Yet another compiler-compiler).

Semantics-based compiler (Peter D. Mosses) Semantics-based compilers are a special kind of compiler-compiler generators that take as input a formal description of the semantics of a language. Peter D. Mosses was the first to produce such a system in 1979, which he called Semantic Implementation System (SIS) [Wan84].

Metaprogramming systems (MPS) [CI84] A metaprogramming system is a programming facility (subprogramming system or language) whose basic data objects include programs and program fragments of some particular programming language, known as the target language of the system. Such systems are designed to facilitate the writing of metaprograms, that is, programs about programs. Metaprograms take as input programs and fragments in a target language, perform various operations on them and possibly generate modified target-language programs. Metaprogramming is a very broad domain including language built-in facilities such as macros, external systems such as OpenC++, compile-time and run-time implementations such as Lisp macros and metaobject protocols. The common denominator is that metaprogramming systems facilities make use, are defined and operate using concepts of general purpose languages.

Software Transformation Systems (STS) [SLB+99] Software transformation systems, or more concisely, transformation systems, are the result

of the merge of symbolic manipulation techniques with compilers. Transformations take software as input and produce software as output. Software transformation systems are tools which are built for such transformations. They range from specific tools for one purpose, via simple pattern matching systems, to general transformation systems which are easily programmed to do any reasonable transformation.

Language Workbenches (Fowler) [**Fow05**] Language Workbenches are a suite of tools such as Intentional Programming, JetBrains's Meta Programming System, and Microsoft's Software Factories. These tools take an old style of development which Fowler calls language-oriented programming and use IDE tooling in a bid to make language-oriented programming a viable approach. One of the strongest qualities of language workbenches is that they alter the relationship between editing and compiling programs. They shift from editing text files to editing the abstract representation of programs. Essentially the promise of language workbenches is that they provide the flexibility of non-embedded DSLs without a semantic barrier between a DSL and its target language.

Language Development Technique

Each of the above carry a connotation, which renders future discussions and analysis subject to interpretation. In order to avoid this, we need to introduce the new term *Language Development Technique* (LDT).

2.2. Definition. *A* ***language development technique*** *is a systematic technique for the implementation of languages without assuming any technology, model, architecture, or a particular kind of semantics.*

We introduce the new term because the existing terminology we discussed does not adhere to this definition for the following reasons: Language workbenches are a class of LDTs which are embedded in IDEs. Those systems actually merge the IDE capabilities with DSLs to lift programming to an abstract interactive and customizable programming environment. Software transformation systems are a specific class of LDTs which implement languages solely via a translation to another language. Metaprogramming systems facilities make use, are defined and operate with concepts of general purpose languages. Language implementations are thus also written in terms of general purpose languages and not in a dedicated system or formalism. Compiler-compilers implement languages through the compilation to a lower-level language or representation and thus excluding optimization, restructuring, or mere translation. Compiler tools form a loosely coupled suite of products each supporting only a specific part of a compiler. In other words, there is no integrated or overall approach underpinning them. Semantics-based compilation is based on the formalization of the meaning of

computer programs by constructing mathematical objects which are independent of the representation and operational semantics of the programming language (abstractness).

2.2.2 Typical Architecture

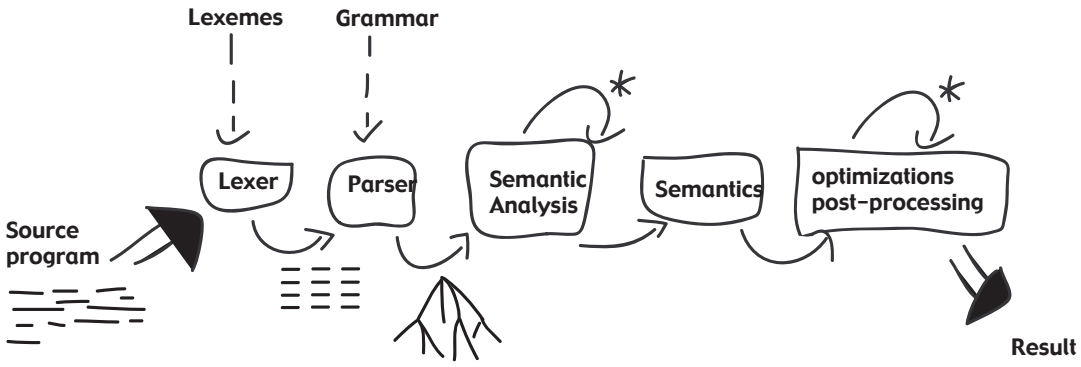

Figure 2.1: A high-level overview of a typical architecture of a LDT.

The typical architecture [SWW+88] (depicted in Figure 2.1) is based on the dominant programming model of the 1960 and 1970ties, named the process-oriented model. Process-oriented structured [FK92] methodologies were developed in the 1970ties for promoting a more effective analysis and design technique based on *structured design and programming* from the 1960ties. Structured programming was based on a series of separate program steps (processes) with respect to data. Process-oriented modules focus on the immaterial[1] flows of data and information. They describe systems as a network of interacting processes, including descriptions of data used by the processes. The architectural style of those systems, reflecting the process-oriented methodology, is a pipeline architecture [GS93].

The subsequent processes of a pipe-line language implementation architecture are called phases or passes. Each phase consumes the input of the previous phase and produces new input or modifies the received one. The first phase consumes the whole program written in the language, the last phase emits the result. As

[1]The process-oriented architectures stem from Eli Whitney's assemblies to create products from parts in a repeatable manner (1739). The flow in these processes were product parts. Hence the emphasis on immaterial information flows when this concept was transfered and generalized by computer scientists.

such, the program gets gradually transformed into a new semantical value or a new syntactical structure.

Lexing and Parsing

The first stage of a language implementation is a lexer. Lexers (also known as scanners) classify a series of input characters into a series of lexical tokens or just tokens. Typical tokens are identifiers, operators, primitive values (strings, bytes, integers, fractions) and keywords (while, begin, function, etc.). Lexers can be formally described via finite state automata. Via these descriptions, lexer generators can produce highly efficient code.

The second stage is occupied by parsers. Parsers implement a grammar specification of a language and is basically able to tell whether an input program is a correct sentence in the language. BNF (Backus-Naur Form) [BBG+60] is the most common notation used for expressing context-free grammars. Usually the goal of a parser is to produce a derivation tree, called a syntax tree. After the parsing phase (or sometimes coincident with the parser), the syntax tree is pruned of all the concrete syntax elements and normalized into an abstract syntax tree. The description of the abstract syntax tree data structure is described via an abstract grammar.

There is a general consensus on the first two phases of a typical language implementation being lexing and parsing. However, that does not imply that every language implementation actually contains those phases. Those two steps are no longer considered absolutely necessary. More so it is argued to be beneficial to discard those phases [Sim95b, Sim96, Jet, Fow05] as it allows a tighter integration with the development environment for managing the vocabulary of DSLs. The two phases are replaced by structure editors which directly manipulate the abstract syntax tree representation of source code. More controversial are the remaining phases such as semantical analysis, code generation and interpretation. A variety of formalisms and techniques have been presented for defining the semantics of programming languages and for the automatic mapping from that description to a complete and fully functional language implementation.

Subsequent phases

The absent consensus on the remaining phases (transformations/interpretation, analysis, optimizations) is mainly due to a large number of formalizations and mechanisms and due to the specific nature (and hence also its challenges) dependent on the particularities of a concrete language implementation at hand such as typing, register allocation, dead-code elimination, etc. The semantics of a language can be expressed in various ways. There are mathematical formalisms such as axiomatic, denotational and operational semantics. Mostly a non-mathematical approach is chosen. These can be divided into interpreters

and compilers/generators. Interpreters implement a denotational semantics in the case of a purely functional language or operational semantics in the case of a procedural language. Compilers implement languages using translational semantics.

2.3. Definition. ***Translational semantics*** *does not define the semantics in the sense that the execution of a language is described directly, but rather preserves its semantics by translating an expression to another language.*

Regardless of the lack of consensus, the functionality in these remaining phases are, to a certain extent, present in many LDTs. The phases are often cleanly subdivided in a strictly ordered series of phases.

The common key operations performed in those phases are all based on information flow. The input of these phases is usually an abstract syntax tree, which is a recursive composition of the abstract representation of syntactical constructs (a.k.a. abstract syntax nodes) bounded by the grammar. Analysis, optimizations and transformations or interpretations often exceed the boundaries of a single abstract syntax node. Hence, mechanisms are required which allow navigation carrying information across various abstract syntax nodes. Well known examples of analysis are type safety, of optimizations are function inlining and constant propagation and of transformations are conditional rewriting. Exactly how these operations are supported, and how these operations should be implemented is a matter of much debate.

An important observation, with respect to the typical architecture, is that these mechanisms offer varying degrees of data descriptions (which are an essential part in pipeline architectures). In contrast to the detailed data descriptions in the lexing and parsing phases by regular expressions and grammars, the data descriptions in the later phases range from well-typed abstract syntax nodes to tree or graph structures. There exists a tension concerning that matter. Firm data descriptions often conflict with flexible implementation mechanisms, while more generic data descriptions conflict with a well designed pipeline architecture.

Conclusion

Clearly the typical architecture is not organized according to the language constructs. Additionally, we observe that the implementation of language constructs is scattered in these phases. It is even unclear how to interpret the various phases as each phase consumes the whole program at some intermediate stage.

Despite the wealth of variation in the phases after lexing and parsing, the common key operations performed in those phases are all based on information flow. The implementation of that information flow among modularized language constructs while maintaining their modularization forms the challenge in this dissertation.

2.2.3 Implementation Approaches

In order to organize the discussion of the various LDTs we subdivided them into five categories according to the degree of support offered for controlling and systematically implementing the various phases of a language implementation. The resulting classification based on these criteria is quite similar to the classifications found in [vDKV00] and [Bri05]. The classification does not assign the systems into distinct categories as the classification lists increasing degrees of control and support.

Ad-hoc Ad-hoc implementations, such as tree traversals [KV01] or transformation tools [BLS98] consist of small tools or libraries which can be used for implementing a single phase of a language implementation. In order to combine the separated phase of a compiler or for implementing a phases on top of some library functionality, one must interface with all the various tools or libraries. This interfacing represents the bulk of a language implementation. It is performed in a general purpose language and requires significant insight in the inner workings of the tools and libraries. In other words, ad-hoc implementations lack an overall systematic approach for implementing languages and more importantly lacks control over the entire language implementation process.

Preset With this kind of approaches, languages are implemented using a fixed set of operations for composing code-fragments. The operations can either be embedded in the target languages or provided by an external language. Typical examples of the former are template languages. In the case the operations are part of an external language, this language acts as a kind of preprocessor. Subject-oriented programming [OKK+96] and Genvoca [BST+94] fall into the latter category. In these systems, a program is an equation of a set of software components or fragments. Because of the fixed vocabulary, the degree of control is limited to the back-end of the language. At most, the vocabulary provides built-in constructs for creating abstractions. These abstractions are the closest you can get to the front-end of languages. So there is no possibility for constructing new language constructs.

Embedded A very popular approach for implementing languages is by embedding them into a general purpose language. Embedded languages come in two forms. The first and most common form offers a (set of) additional language construct(s) which allows you for expanding a syntactical expression by another one. A prime example of this subcategory are macros. The syntactical latitude and the abilities for expanding code varies from one implementation to the next. C macros only support two syntactical forms, and are implemented by simple text substitutions. Lisp macros reside at the

other end of the spectrum and are the most powerful macro implementations available. The second form of embedded languages are constructed by exploiting the flexibility available in a host language. Existing implementations range from functional binding, lambda abstractions, higher order functions, type system extensions, to metaprotocols. Embedding lacks syntactical control, when the syntax of a new language does not match the syntax of the host language. But more importantly, its grammar cannot be described in a declarative way and thus cannot be easily enforced. The code fragments of a new language are intermingled with host language fragments. Consequently, embedded DSL programming is more prone to errors, than an interpreted or compiled language.

Interpretation Interpreters are often easier to write than compilers. Both the front-end and back-end of a language can fairly easy be implemented with no profound restrictions. The back-end of a language consists of its operational semantics. The operational semantics for a programming language defines the meaning of a valid program in terms of how a program is processed via sequences of computational steps. These sequences then are the meaning of the program. In a functional style, an interpreter is basically a recursive function over the abstract syntax representation of a program. Interpreters do not allow post factum changes to be made to previously computed values. So, although all the phases of a language can be implemented in an interpreter approach, the typical order of the phases of a language implementation has to be slightly changed in the case optimizations have to be applied and execution dependencies have to be resolved. Although, JIT [Ayc03] compilation is said to be used to optimize interpreters, this technique does not invalidate the previous statement. The goal of a JIT compiler is to combine many of the advantages of native and bytecode compilation: expensive computations such as parsing the original source code and performing basic optimization are handled at compile time. Hence, prior to interpretation a compilation phase is used to implement the basic optimizations. Upon the execution of the bytecodes, the bytecodes are translated to machine code. Again this process is more like a compiler instead of an interpreter.

Compilation Compilation is the most advanced language implementation approach available. Developers have full control over the implementation of a language. No concessions have to be made regarding the structure of a language implementation. Every language phase can be executed whenever the input data for that phase is available and the phases can be subdivided such that every phase only concerns itself with a particular task. The main distinct enabler in compilers for this, in contrast with interpreters, is that compilers preserve the semantics by producing an equivalent program.

We refer to this as translational semantics (Definition 2.3). The equivalent program of a given source program gets gradually produced and/or refined as the phases progress. Hence, compilers allow various phases to further process the code before execution. As such, the implementation is more comprehensible and more maintainable in terms of its tasks.

Although we primarily focus on compilation in this dissertation, there are two reasons to include other systems as well. First, each of them has some benefits and bring about some important insights appropriate for our work. Second, compilation is the superset of other generative approaches. Therefore they are interesting for their particular aspect of the compilation process. The ad-hoc approaches have diverse strengths, depending on which particular part of a language implementation process they facilitate, e.g. a symbol table or a traversal library for exchanging information in the compilation process. The preset approach is known for its techniques for composing various software components and fragments. The embedded approaches are known for the combination of syntactical and semantical semantics of a language construct. In the next chapter, LDTs in each of those categories are discussed.

In the next section we discuss the characteristics of compilation based approaches which impact the modularization of language constructs.

2.3 Translational Semantics

In this dissertation we consider compilation as an approach for language implementations, which includes besides the pure compilation approaches also the embedded, preset and ad-hoc approaches. The semantic phase of each compilation-based LDT consists of a set of modules which contain the translational semantics for a particular source code fragment. The terminology used to denote these modules is different in each kind of LDT as it is tied to a particular technology, paradigm or formalism. To be able to cross these technological barriers we refer to these models as transformation modules or simply transformations.

2.4. Definition. *In its most general form, a* ***transformation module*** *consumes a number of code fragments and produces a number of fragments. Depending on the LDT at hand, modules are restricted to consume one code fragment, to consume several source language program fragments, to produce new fragments, to change existing fragments, to produce only a single fragment, or to produce several target language program fragments.*

To get a better understanding of the challenges ahead, we investigate the impact of the various features or characteristics of transformation systems on the attempt to divide the transformation process into a set of modularized transformation modules. There are many features or characteristics of transformation

systems[2]. There are two characteristics which have a direct impact on the division into modules and these are *granularity* and *scope*. The direction of a transformation is significant as transformations rely on various mechanisms depending which view is taken. However, as will be shown in this section, when considering fine-grained transformation modules the direction becomes unimportant. The complications due to the fine-grained transformation modules and the various scopes are investigated in this dissertation.

The next three subsections discuss granularity, direction and scope.

2.3.1 Granularity of Transformation Modules

When writing a set of transformation modules, decisions on the granularity of those modules must be made. The granularity of a transformation module is the portion or pivot of a source tree which is transformed, ranging from a single AST node (small granularity), or a large subtree of AST nodes (large granularity) which are all transformed at once.

Let us illustrate the impact of granularity by implementing a DSL compiler with rule-based transformation systems (see Section 3.1). Basically a rewrite rule matches a code fragment in its left hand side and substitutes that code fragment by a new fragment described in its right hand side.

Consider the following abstract syntax tree (AST) of an example program which describes a multimedia article of a newspaper. The tree contains one subject which contains an single article and a corresponding title. A title has two parts, a main title and a subtitle.

```
subject(title("first title", "second title"),
        article("The new council abruptly canceled ...")
```

Suppose we want to compile this AST into a HTML page. Titles must be centered and printed above each other, centered at the top of the page. Subtitles must be printed in gray and in font size 9.

As it is our intention to modularize language implementations along their language constructs, it is clear that we opt for fine-grained transformation modules where we divide the transformation process into small rewrite rules that capture the smallest possible structure which they still can rewrite. This strategy maximizes the amount of reusable rewrite logic and minimizes the need to split up existing rewrite rules. Any increase in variability requires a minimum amount

[2]Classifications capturing all the features or characteristics rapidly result in a multi-dimensional blob describing syntactic separation between left hand side and right hand side, bidirectionality, parameterization, typing of metavariables, patterns, granularity, scope, direction, source-target-relationship (new target, in-place destructive or in-place update), implicit vs explicit rule invocation, staging or phases, automation, intention, paradigm, goal, and many more [SD02, CH03, vWV03]

of implementation effort because of the reuse possibilities and avoids constant refactoring of the existing rewrite rules.

The above example program contained two concepts: a title and a subject. The implementation of its transformation yields (see below) a rewrite rule for subjects (1), a rewrite rule for articles (2), and a rewrite rule for titles (3). Titles are rewritten into a html structure rendering the main and the subtitle centered, in the requested font and above each other. Since the subject merely contains a title, subjects are rewritten as its title with a subtitle.

```
subject(title, article) = title br article

article(text) = text

title(text,text2) = center(text br font(text2, 9, "gray"))
```

The following two small evolutions can easily be incorporated in this implementation of our example language. Adding articles to the subjects merely involves changing the subject rewrite rule to incorporate more then one article (1) and adding a variant of text requires only the addition of one new title rule (2).

```
subject(title articles*) = title br articles*

title(text) = center(text)
```

Although this fine-grained division of a transformation into several rewrite rules works perfectly well in this example, it also complicates matters. Firstly, the smaller the code fragment covered by a rule is, the broader the scope that rewrite rule can apply to. Hence, rules need to be *additionally scoped and scheduled*. Secondly, the smaller the input or output code fragment is, the less information is available for executing the transformation. So *additional behavior is necessary for compensating* this loss of information. Thirdly, the smaller the code fragments get, the more rewrite rules will depend on the results of other rules. A rule which transforms a large code fragment, can simply transform the whole code fragment at once and can produce a whole target fragment. As rules consume smaller code fragments, then each of the code fragments produced by smaller rules have to be combined to yield a whole target fragment. As such, rules depend on the results of other rules. These *additional dependencies* were absent in a coarse-grained modularization.

2.3.2 Direction of Transformations

The two mainly acknowledged and used directions in the implementation of transformations are target-driven approaches and source-driven approaches.

In a *target-driven approach* (depicted in Figure 2.2 (a)), also called goal-driven approach, the target program structure steers the execution of the transformation. Transformation modules are responsible for a particular target program fragment that needs to be created. As such, they produce and collect all the parts that are needed for completing a target program fragment. Target-driven LDTs are equipped with powerful query mechanisms to be able to compute and collect the needed information and parts across a source program.

In a *source-driven approach* (depicted in Figure 2.2 (b)), also called data-driven approach, the source program structure steers the execution of the transformation process. Source programs are the data which triggers the transformation modules that need to be executed. A transformation module contains the translational semantics of a source fragment and computes the information necessary for executing its semantics. The amount of control determined by source programs varies among LDTs. In its most basic form, target programs are constructed by traversing source programs and applying the correct transformations.

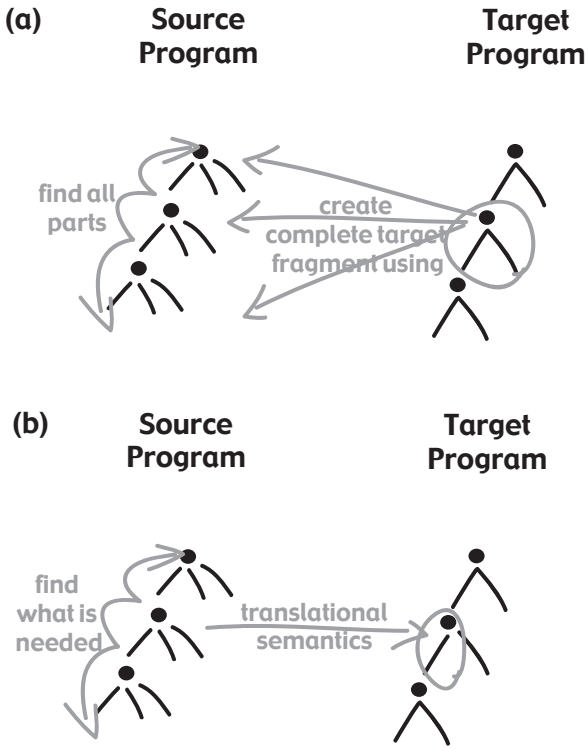

Figure 2.2: The direction of a transformation: (a) a schematic example of a target-driven approach (b) a schematic example of a source-driven approach

In the next section we show that the direction of transformations changes how their scope affects the division of the transformation process into fine-grained

modules, but does not change the essence of the impact of the scope.

2.3.3 Scope of Transformations

Fine-grained transformation modules transform only a small part of a source program. As the amount of input is small, they produce only a small part of a target program. Another characteristic of a transformation model, which is related but not to be confused with the granularity of transformation module, is scope.

The *scope* [vWV03] of a transformation module are the areas of a source and target tree taken into account by a transformation module. It concerns the areas of a program that are affected by a transformation and from which information is used. A *local scope* covers a single portion or *pivot* of a tree. A local scope can have a small or a large granularity, e.g. a single AST node or a large AST subtree respectively. We distinguish between two kinds of scopes: local and global scopes. A *local scope* covers a single subtree of the program. When there is more than one disconnected subtree covered, the scope is *global*.

Van Wijngaarden [vWV03] classifies transformations by their input and output scope. The *input scope* or *source scope* denotes the area of the source program which is covered by the transformation. The *output scope* or *target scope* denotes the area of the target program affected or produced by the transformation.

Joining the axis of source and target scopes with local and global scopes we get the following transformation scopes:

- A *local source scope* takes a single portion or *pivot* of a source tree.
- A *local target scope* comprises a single portion or *pivot* of a target tree.
- A *global source scope* takes into account several additional source nodes next to the pivot or portion of a source tree under transformation. These nodes are not transformed by the transformation module, but merely necessary as an additional information source.
- A *global target scope* comprises several portions or areas of a target tree which are affected by a transformation module.

Joining the axis of source and target scopes each divided into a local and global scope we get the four possible transformations listed in Table 2.1.

input/output	local	global
local	local-to-local	local-to-global
global	global-to-local	global-to-global

Table 2.1: Classification of source and target scopes of transformation modules

Figure 2.3 illustrates these basic terms in a schematical overview of a transformation. The main input of a transformation is referred to as the pivot. The main output of a transformation is called *local result*. Information required by a transformation which is external to the pivot is called *external* or *context information*. The extra results produces besides the main or local result are called *nonlocal results*.

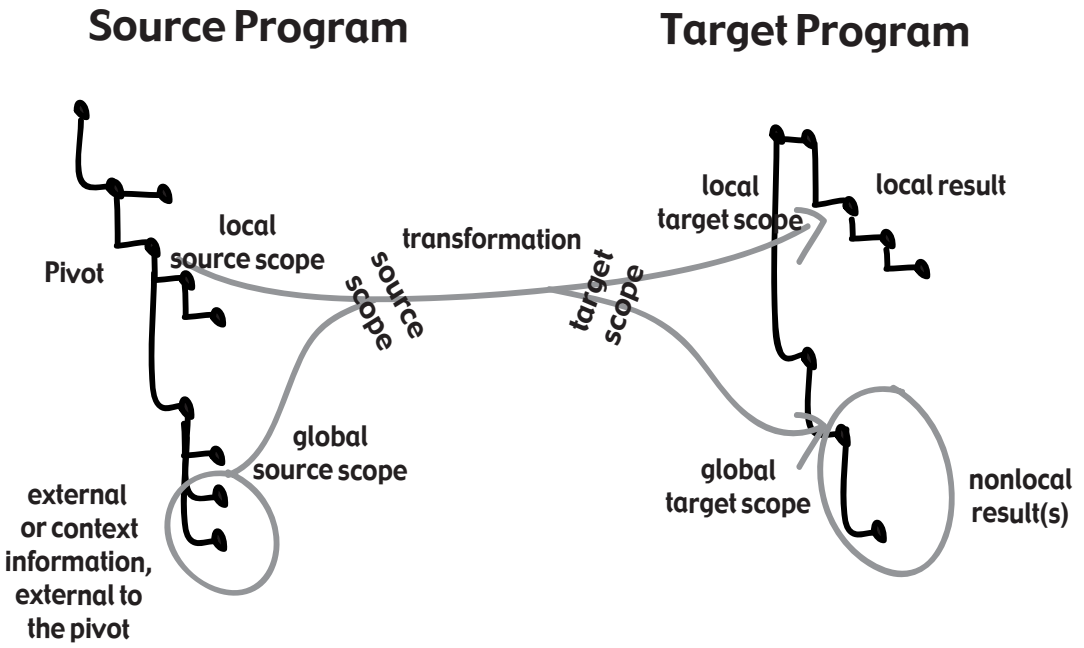

Figure 2.3: A schematical overview of transformation terminology.

In the remainder of this subsection we discuss the various kind of scopes of transformation modules.

Local-to-Local Transformations

The local-to-local transformations are the most simple kinds of transformations for constructing and for supporting the LDTs. The local input and local output renders the transformations largely independent from one another. The only dependencies that need to be taken into account are those among the transformations where one operates on a parent and the others on the children nodes. The dependency lies in the fact that the results produced by transforming the children must be composable in the result produced by transforming the parent.

In the case of a source-driven approach, the basic execution scheme boils down to a recursive descent bottom-up traversal of the source program. In the case of

a target approach the execution scheme boils down to a recursive descent top-down traversal of the target program. The prerequisite for these simple schemes to work, is that *the information in the local scope must suffice for constructing a locally scoped target term.* That prerequisite erects a dependency between the transformation modules which must be taken care of by the language designers. Although the dependency is of a different nature in a source or target driven approach, but nevertheless present in both of them. So, the choice between a source and target driven approach does not affect the complexity of the implementation of these transformations.

- In a source-driven approach, the source program is traversed, dictating the target scopes (see Figure 2.5) i.e. the kind of output that is allowed to be produced by a transformation and the location of the produced code is determined so that it is composable with the results produced by other transformations.

  The goal of each transformation T_A is the production of the translation semantics A' of the source term A. This transformation uses the semantics B' of the subterms B by invoking the transformations T_B. These values are used for constructing the target term A'. The results B' obtained from T_B must be suitable values for constructing the target term A'. So a source-driven transformation dictates the target scope of the produced target program fragments.

- In a target-driven approach, the target program is traversed, dictating the source scopes (see Figure 2.4) i.e. the input for a transformation is determined and must suffice to produce the results.

  The goal of each transformation T_A is to produce a complete target term A', using the source term A. In order to construct the target term A' a number of subterms B' of A' are necessary. The transformations T_B producing these subterms are invoked with a part B of the source term. That part is determined by the transformation T_A and has to be suitable for the transformation modules T_B. So a target-driven transformation dictates the source scope.

Global-to-Global Transformations

The remaining three classes of transformations access multiple disconnected source subtrees and/or affect/produce multiple disconnected subtrees in a target program. These transformations discord with the goal of fine-grained modules. In designing a language with fine-grained transformations we strive to limit the scope

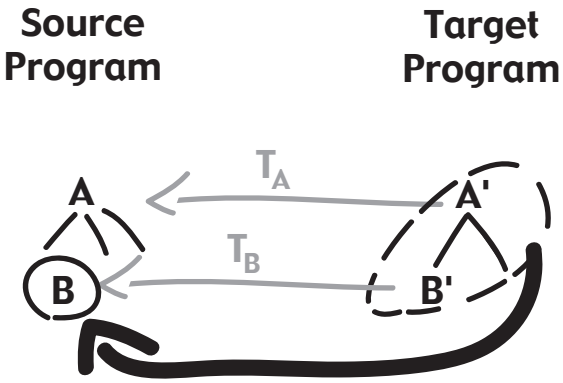

Figure 2.4: A target-driven implementation of local-to-local transformations dictates the scope of the source of the transformations.

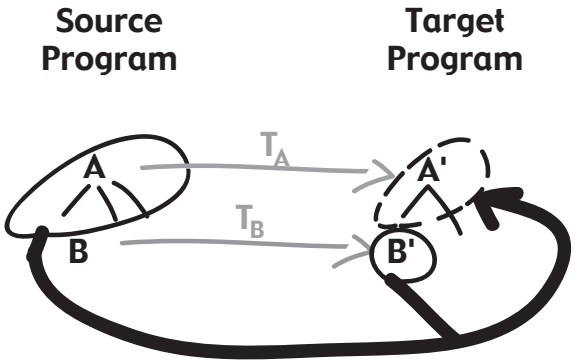

Figure 2.5: A Source-driven implementation of local-to-local transformations dictates the scope of the target of the transformations.

of transformations as much as possible, whereas we now see that transformations may equally well depend on the whole input program and affect the whole output program. To this end, LDTs offer special support in terms of implementation mechanisms and techniques for facilitating the implementation of these transformations by fine-grained transformation modules. We discuss these techniques in the next chapter.

Global-to-global transformations have a global source scope and a global target scope. They are the combination of the two classes global-to-local and local-to-global transformations capturing a global source scope and global target scope respectively. By dividing the global-to-global category up we can consider each case separately from the other.

Global-to-local transformations need information from multiple disconnected subtrees of a source program. Usually, one assigns different roles to the subtrees. The main subtree which is the trigger of the transformation is the pivot [vWV03] (see Figure 2.3). The other subtrees are considered as *context* information (see Figure 2.3). A typical example of such transformations are those that compile variable usage sites in programs. A variable usage corresponds to a memory address that was allocated upon its declaration. So for translating a variable to a memory address its transformation needs to lookup its declaration for retrieving the memory address.

As this is a common transformation, many LDT explicitly offer additional mechanisms for supporting the retrieval of information ranging from code walks to global accessible data structures. LDTs each provide their own mechanisms and techniques. Hence, we will come back to this during the more detailed discussion of LDTs in chapter 3

Local-to-global transformations produce/affect several disconnected subtrees (aka results) in a target program. In order to implement these transformations, nonlocal results must be spread across the overall target program and integrated with the existing parts of a target program. Clearly, scheduling is an important issue here, as such a transformation can only be completely executed when those scattered locations in a target program can be computed. Only a couple of transformation techniques provide special support for local-to-global transformations. But most of them treat the problem of local-to-global transformation as the inverse problem of global-to-local transformations. From a technical point of view, this is correct; spreading the results throughout a target program can be achieved by looking for the results from the subtrees where they belong. Like the global-to-local transformations, the mechanisms and techniques for implementing local-to-global transformations are part of the detailed discussion of each LDTs. However, as we will discuss later on there are several problems associated with the inversion solution.

The implementation of local-to-global transformation is performed in target-driven approaches by querying a source program and invoking the right transformation modules for obtaining target program fragments. Subsequently, these fragments need to be composed and integrated into a single target program. In source-driven approaches, a target program is queried for locating where to integrate nonlocal results. Subsequently, nonlocals are integrated in the target program. Clearly, local-to-global transformations require the same logic in both target or source-driven approaches.

2.3.4 Discussion

Despite the technical differences used in target-driven approaches and source-driven approaches, target-driven approaches do not cancel disadvantages of source-driven approaches or vice versa. The implementation of local-to-local transformations, global-to-local transformations and local-to-global transformations is largely the same in both source and target-driven approaches.

2.4 Conclusion

Growing a language can have a significant impact on the semantics of other language constructs, as the addition of an expressive language construct requires changes to be made in the definition of other language constructs. We have shown that a set of language constructs does not in general compose, unless they are designed to compose or they are complemented with additional logic.

We have given an overview of the suite of contemporary language development techniques of which we will, in later chapters, determine the extent to what degree modularization can be achieved.

There is a wealth of names given to techniques, systems and formalisms for implementing languages. Each of those carry a connotation, which renders future discussions and analysis subjective to interpretation. In order to avoid this, we introduced the new term "Language Development Technique (LDT)".

To get a better understanding of the challenges ahead, we analyzed the granularity, scope and the direction of a transformation. Granularity and scope are two characteristics which have a direct impact. The direction of a transformation is usually considered significant, however, we showed that when considering fine-grained transformation modules the direction becomes irrelevant.

In the next chapter, we present the mechanisms each individual contemporary language development technique offers for modularizing their implementations and for handling the various scopes of transformations.

Chapter 3

Language Development Techniques

In the previous chapter, we have introduced basic terminology and approaches to implement languages. In this chapter, we present a detailed study of each language development technique. The existing research tools we are about to discuss is extensive, including tree-based rewrite rule systems, graph rewrite rules, macros, template-based approaches, attribute grammars, compositional generators and ad-hoc approaches.

The reason why we need to cover so much ground is due to the fact that we pursue modularization through an appropriate separation of concerns. Separation of concerns [Par72] is a highly desirable property in software engineering in general and in language engineering [Mos04] in particular. It is a concept that is not a clear-cut property. Therefore each language development technique (LDT) (see Definition 2.2) divides a language implementation into modules ensuring some degree of separation of concerns according to some separation of concerns dimension [TOHJ99].

The various phases in the typical compiler architecture (see Section 2.2.2) such as lexer, grammar, parser, transformations, optimizations, pretty printing, language constructs, analysis, etc. are the main sources for concerns. Many of these, are further refined into specific concerns due to the particularities of the source and target language such as typing, register allocation and dead code elimination. In addition, compiler construction is a very diverse field, governed by several compiler construction paradigms such as rewriting and attribute grammars. Each paradigm abstracts and structures the compiler construction process slightly different. As a result, the various technical concerns that explicitly arise during compiler construction are partially dependent on that paradigm.

Our discussion focuses on the degree of separation that can be achieved by contemporary development techniques. In Section 2.3.1, we have outlined the complications due to the fine-grained division of a transformation. The more fine-grained a module is, the smaller its source and target scope gets. As a consequence, transformations need additional scoping and scheduling. In addition,

additional behavior is necessary for compensating the reduced amount of information available for executing the transformation. In order to chart the impact on the modularization of transformation modules, we detail the cohesion of the various modules, how they cooperate with other modules, and their dependencies and coupling with other modules.

Each LDT, provides useful insights in how to divide a language implementation and what mechanisms are necessary to maintain that separation. We present the mechanisms to separate as well as the mechanisms to use and combine the modules into a fully functional language implementation. Each mechanism and property relevant for our discussion is briefly introduced together with its strengths and weaknesses. These mechanisms and the properties of the systems form the background knowledge we use and take into consideration in the next chapter. That chapter presents a model for the modularization of language constructs and a summary of a thorough evaluation of the mechanisms of each LDT against that model.

The discussion of each LDT is structured as follows:

- The definition and properties of the data structures that are used in each LDT are important as they define the scope for the possible operations to transform and manipulate the program representation. We take a look at all the data structures that are used in a language implementation. This includes the source program, the target program, and the auxiliary data structures. We focus on:
  - The structure and organization of source and target programs: tree, graph, changeable or non-changeable data structures.
  - The construction of target programs and auxiliary data structures: implicit, explicit, complete or incomplete.
  - The modification of these data structures: unconstrained vs constrained and ownership.
- We determine the smallest possible semantic operation that can be captured in a single transformation and determine how the various scopes are handled, so as establish the degree of fine-grained modularization. In order to clarify the discussion of each LDT, we graphically depict the transformation modules they offer. The drawing also depicts the main properties and mechanisms of each LDT, including their scope and access to source and target trees. The drawing is based on a common illustration of the various scopes of a transformation in Figure 2.3. So each figure shares the same concepts and drawings, providing a common platform to compare the various approaches. We investigate how:
  - modules identify what to transform and the scope of that transformation;

  - transformations are scheduled;
  - local-to-local transformations are expressed;
  - local-to-global transformations are expressed;
  - global-to-local transformations are expressed.

- We conclude each discussion with its strengths and weaknesses. The former are indicated with a '+' sign and the latter with a '−' sign.

Note that we refrain from a complete concrete technical overview of all the systems currently available. The list of LDTs we are about to discuss is a representative set of techniques to support the modularization of language constructs and their semantics. In addition, the focus of our discussion lies on the techniques LDTs offer to divide an implementation into modules according to the language constructs of a language, and the techniques to preserve that modularization when facing complex translational semantics. Our discussion is thus conducted from that perspective only and should be read accordingly, it does not contain a full evaluation including pros and cons of every feature offered by the LDTs. We conclude with a couple of observations governing the field of language implementations through compilation.

3.1 Tree-Based Rewrite Rules

Rewrite rule systems [Klo92] such as ASF+SDF [vDHK96], TXL [CDMS01] and Stratego [Vis01b] can be counted as one of the most flexible formalisms for expressing program transformations. This dissertation covers rewrite rule systems for three reasons: First, rewrite rules are fine-grained and highly modular. Second, the ASF+SDF rewrite rule system offers traversals, which is a mechanism to implement local-to-global transformations. Third, stratego allows developers to explicitly control the scheduling of rewrite rules and to dynamically create rewrite rules which can implement local-to-global transformations.

A rewrite rule (see Figure 3.3) basically substitutes a pattern of a code fragment for another pattern, called the left hand side (LHS) and the right hand side (RHS) of a rule respectively.

Consider the simple abstract tree language defined in Figure 3.1. The binary trees of this language consist of three different node types `f`, `g` and `h`. The leaves are just plain numbers (`NAT`). This specific example happens to be a complete binary tree, which means all nodes have exactly zero or two child nodes. Transforming all the `f`-nodes into `h`-nodes can be done by a simple rewrite rule shown in Figure 3.2. The left-hand side matches every `f`-node. These nodes are rewritten to `h`-nodes.

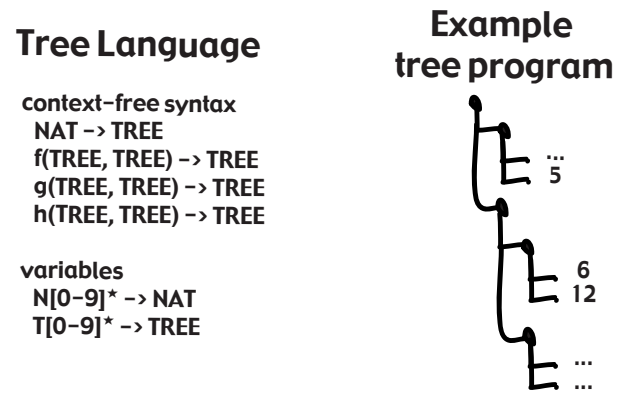

Figure 3.1: On the left side a simple tree language is defined, on the right side an example tree that can be constructed using that language is depicted.

```
f(T1, T2) = h(T1, T2)
```

Figure 3.2: An example of a rewrite rule

3.1.1 Data Structures

Structure and Organization The program under transformation is represented with an abstract tree, for example `f(f(g(1,2),3),4)`. The trees are constructed by composing terms with other terms and basic elements.

The tree structure is only captured by parent-children relations. This has important consequences for the acquisition of information (see Section 3.1.2).

There is at all times, during the transformation process, a single set of connected terms available. At the beginning of the transformation process, the terms represent the source program. As the transformation progresses, the terms are gradually rewritten until the terms represent the target program. So, during the transformation process an intermediate tree intermingling source and target program fragments exists.

The rewrite rules are used to implement the whole back-end of a compiler. In other words, they are used for semantical analysis, translational semantics and optimizations.

Construction The construction of a target program is explicitly stated in the right hand side of rewrite rules. Every constructed term must also be complete and correct. The correctness is enforced by a type system and through the term constructors. The terms are constructed by composing all its required subterms.

Modification Terms are read-only. Their content can only be changed by recreating them with different subterms as parameters. If their type and their arguments are not sufficient, terms have to be recreated. *A rewrite rule strictly depends on the signature of a term*, when the signature of a term changes (see local-to-global) the rewrite rule is no longer is applicable.

3.1.2 Transformation Modules

A transformation module is a rewrite rule. Rewrite rules can be organized into groups of rewrite rules. The transformation modules or groups of modules are not organized according the structure of the source language defined by its grammar.

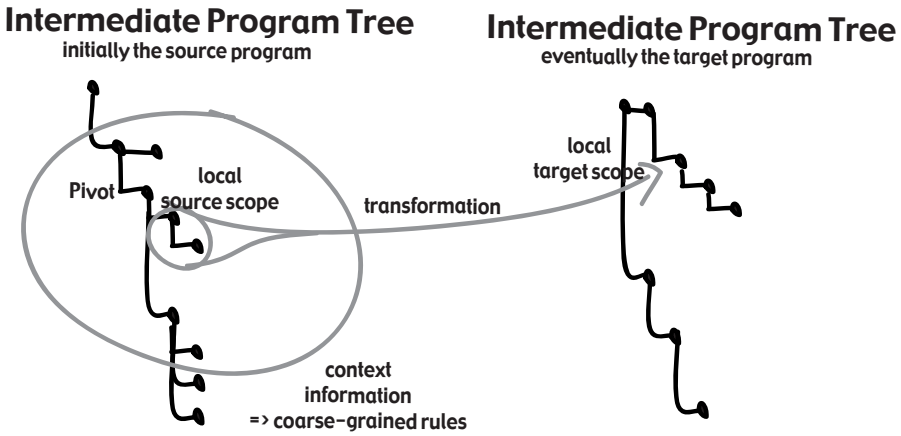

Figure 3.3: A schematic overview of a rewrite rule

Identification The left hand side pattern of a rewrite rule is, in its most basic form, a term. These terms can be arbitrarily expanded by their subterms to allow for more complex patterns. However, this simple form of expansion, as presented here, has its limitations. As a term must always be fully expanded – there is no room to abstract over a number of concrete terms or, in other words, to enable a certain degree of variability. This variability is achieved by the addition of variables. Such terms are then called patterns.

The patterns available in rewrite rule paradigms are matched against the current tree under transformation. The variables in its left hand side act like input parameters and in its right hand side like output parameters.

Variables enable only a limited form of variability as the structure of the pattern remains statically fixed. As rewrite rules depend on the structure of the tree, which may in turn be rewritten by other rules, *rewrite rules implicitly depend on each other*. Extensions of the basic rewrite rule formalism discussed

in Section 3.1.3 and related approaches such as graph rewrite rules in Section 3.2 offer solutions with a varying degree of variability.

Scope The source scope of a rewrite rule is part of its definition. The scope is determined by an additional condition attached to a rule and in its left hand side. The left-hand side (LHS) statically binds the scope to a particular term whereas conditions further specializes that and can take dynamic information into account. As terms lack a child-parent relationship, a condition only applies to the subterms of the matched term. Therefore, the term described in a LHS must be a large enough subtree, such that conditions can be defined in terms of its subtrees. So despite of this very modular and localized semantics, *rules are tightly coupled with their overall language context.*

Scheduling In general, and conceptually, the process of term rewriting continues until no further rules can be applied (or for ever, in the case of a non-terminating system). For rewrite rule systems to be reliable, the set of rewrite rules must be *confluent*. When we reach a point where we can apply no more rewrite rules, we have reached a *normal form*. If there is only one unique normal form, then our system of rewrite rules is confluent. In general, determining if a rewrite system is terminating and/or confluent is undecidable: fortunately there are some useful rules to help us decide.

Changing a non-confluent system to a confluent system is done through proper rule scheduling. Scheduling of rewrite rules is based on a built-in strategy determined by the reduction strategy, that is, the procedure used to select a subterm for possible reduction. In ASF+SDF, the leftmost-innermost reduction strategy is used. This means that a left-to-right, depth-first traversal of the term is performed and that for each subterm encountered an attempt is made to reduce it. Built-in reduction strategies have their limitations. To overcome these, the conditions of rewrite rules can also be used to explicitly control their execution. *The conditional clause is a part of the definition of a rewrite rule, rendering it dependent on its context [Cle03].*

Rewriting complicates scheduling because rewrites may remove information and rewrites can change information. Before a rewrite rule can be safely applied, we must make sure that all other rewrite rules which also depend on that system are either executed or make a copy of the information that is about to be removed. Before a rewrite can be safely applied, we must make sure that previous computations based on that information are not invalidated. *Enforcing a schedule in these systems is done by changing the conditional clauses of rewrite rules (cfr. the byte code generator example in [vdBK02]).*

Effect Rewrite rules can only affect the matched term by replacing it with its resulting term. In other words, it can only affect the term that is matched by

the left hand side, other terms cannot be affected. Hence, the target scope of a rewrite rule (its right hand side) is restricted to the source scope (its left hand side). In other words, rewrite rules can only affect a single subtree at a time using information of that subtree. *Consequently, transformations with global source and/or target scopes are not explicitly supported.*

Global-to-Local Due to the absence of child-parent relationships, the minimal tree representation prohibits a straightforward way of accessing terms residing in the context of a matched term. So the only way to be able to access context information would be by matching its parent terms. *The increased scope of the resulting rewrite rules render them coarse-grained* (cfr. Section 2.3.1).

Technically speaking, rewrite rules can access context information through the mechanism of successive rewrites. The idea is to distribute information to the terms by rewriting each of the encountered terms along the path where the information is computed and made available to the term where the information is required. This is not only *cumbersome but also causes severe maintenance penalties* [CDMS02].

Local-to-Global As rewrite rules can only exhibit changes to the matched subtree, *implementing a local-to-global transformation is a hassle.* The only way to affect other subtrees is to gradually rewrite the tree until sufficient information is available at those subtrees that need to be changed. There are two approaches which can be taken: restructure the source tree or restructure the target tree.

In the first approach, the source-tree is restructured until there is enough information available in the intermediate "decorated" source-tree to be able to execute the rewrite rules that actually produce the target program. The drawback of this approach is that *the logic of the local-to-global transformation is scattered in the rewrite rules that actually produce the target program.* In [CDMS02], this approach is "successfully" applied in the large to transform legacy Cobol programs to Java programs. More than 70 intermediate stages are needed before a correct Java program is reached.

In the other approach, first the rewrite rules that actually produce the target program are executed. These rewrite rules produce an intermediate tree which is "decorated" with nonlocal subtrees. In subsequent phases this intermediate target tree is rewritten. Every successive rewrite restructures the tree and moves nonlocal subtrees upwards or downwards in the tree, until the nonlocal subtrees reach the subtrees in which they need to become a part of. As the target tree is usually more verbose, less concise and less intentional (lower-level language), the drawback of this approach is that *the scattering of the subtrees can be more complex to implement.* We investigated this approach in [Cle05, CB05] and the major critiques are the lack of integration semantics.

Strengths and Weaknesses

- Rewrite rules are highly modular. It can only utilize the information stored in a term and change the term that has been matched. But this modularity has its price. Local-to-global transformations and global-to-local transformation are not explicitly supported and are cumbersome to implement.

- Rewrite rules are destructive. One must ensure that in case of global-to-local transformations, rewrite rules that depend on external information are executed first. Hence, rewrite rules increase the need to schedule.

- Rewrite rules do not distinguish between source and target programs. That complicates reasoning and typing as intermediate trees may contain both source and target program fragments.

- Rewrite rules can change every term, which may invalidate previously computed information.

- Rewrite rules can only create complete terms. So prior to rewriting, all necessary information must be present.

3.1.3 Traversals

Recall that the patterns in the left hand side of rewrite rules are partially expanded terms. This expansion is statically described by nesting terms. The variables in these patterns enable a certain degree of variability but they actually only confine the expansion. An important consequence is that the structure of patterns is statically fixed. As the program tree is continuously rewritten, *the patterns erect strong dependencies*. In other words, there is no room for structural variability in patterns. With the advent of traversals, this shortcoming can be tackled. Traversals (see Figure 3.5) have been added to the rewrite rule paradigm to alleviate the programmer of the cumbersome programming needed to distribute context information via successive rewriting. The techniques proposed in [VKV03] allow a rewrite rule to descend into a subtree. During the descent, information can be accumulated and/or nodes can be rewritten. The advantage of that approach is that the terms that need to be visited can be formulated as a normal rewrite rule.

Consider the following example where we need to increment the values of a tree given a certain increment. The tree is defined in Figure 3.1. The implementation is shown in Figure 3.4. The traversal function `inc` increments every number in the given tree `Tree` with a given number `N2`. The traversal is declared by an additional specification along with the definition of the term `inc`.

```
context-free syntax
   inc(TREE, NAT) -> TREE { traversal(trafo,bottom-up,continue) }
equations
   inc(N1, N2) = N1 + N2
```

Figure 3.4: Incrementing the leaves of a tree. For example, `inc( f(g(1,2), 3), 10)` results in the tree `f(g(11,12), 13)`.

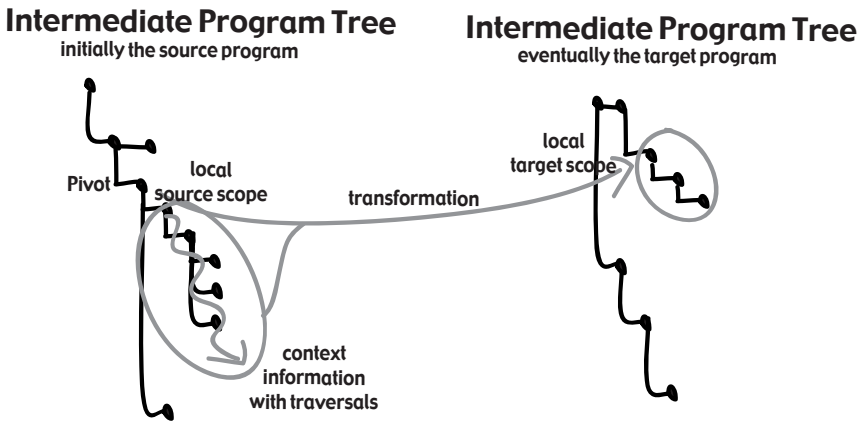

Figure 3.5: A schematic overview of a rewrite rule equipped with traversals

Scope The target scope of a rewrite rule with traversals is no longer confined to the matched terms, more distant subterms (subterms of subterms and beyond) can be reached and rewritten.

Local-to-Global and Global-to-Local Global-to-local transformations need context information which is not locally available in a term. That additional information can now be provided by traversals. *Consequently, only transformations with global source scopes confined to subtrees and with local scope are explicitly supported.*

Traversals are extended over tree such that it distributes the data needed by transformation rules. The disadvantage of these solutions is that the traversal strategy becomes data heavy instead of just handling control flow. That is, *all traversal functions become infected with additional parameters carrying context information.*

Ancestors and siblings cannot be elegantly retrieved. In these cases, a workaround must still be used which involves the rewriting of at least the term

that requires the context information, combined with two or more traversals starting at a common ancestor. *The additional rewrite rules necessary to implement the workaround implicitly cooperate with one another.*

Traversals do change how global-to-local transformations can be implemented. Complete terms can now more easily be created as information can be retrieved from subtrees. The traversals partially eliminate the severe technical and maintainability hazards due to term rewriting. The required information to construct a term can now be more easily computed and retrieved from subtrees. Hence, the number of incremental representations can be greatly reduced.

Strengths and Weaknesses

\+ Traversals can collect information from subtrees to implement local-to-global transformations, and as such reduce dependencies caused by successive rewriting.

\+ Traversals are type-safe.

\- Traversals do not allow parent access and are thus limited to subtrees.

\+ Traversals reduce the number of incremental representations when implementing global-to-local transformations, and thus reduce dependencies among rewrite rules.

3.1.4 Scoped Dynamic Rewrite Rules with Rewrite Strategies

Scoped dynamic rewrite rules [Vis01a] (see Figure 3.6) are an extension of plain rewrite rule systems which facilitate the implementation of local-to-global transformations. In order to control the application of these rules, the technique is embedded in programmable rewriting strategies. With programmable rewriting strategies the application of rules can be controlled/scheduled. The scoped dynamic rewrite rules offer a mechanism to create and revoke during the execution of the transformation process new rules. By creating such a rule, a rewrite rule can have a global target scope.

Scheduling Exhaustive application of all rules to the entire abstract syntax tree of a program is not adequate for most transformation problems as the system of rewrite rules expressing basic transformations is often non-confluent and/or non-terminating.

An ad hoc solution, that is often used, is to encode control over the application of rules into the rules themselves by introducing additional function symbols. This intertwining of rules and strategy obscures the underlying program equalities. It also incurs a programming penalty in the form of rules that define a traversal

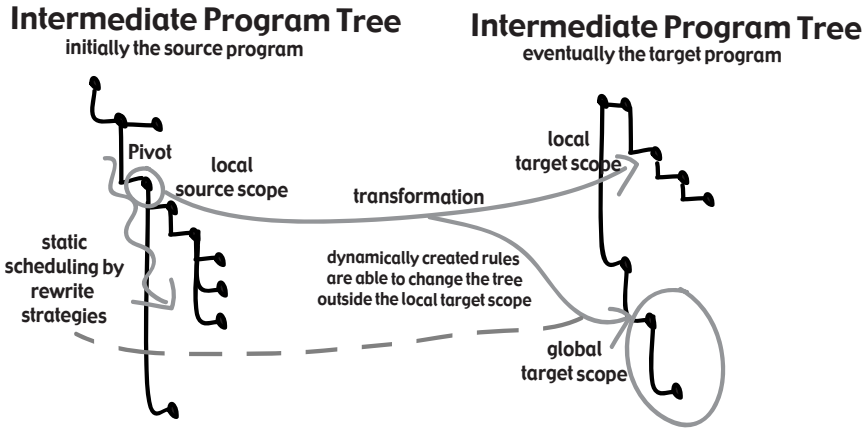

Figure 3.6: A schematic overview of a dynamically scoped rewrite rule

through the abstract syntax tree, and disables the reuse of rules in different transformations.

The paradigm of programmable rewriting strategies solves the problem of control over the application of rules while maintaining the separation of rules and strategies. A strategy is a little program that makes a selection from the available rules and defines the order and position in the tree for applying the rules. Thus rules remain pure, are not intertwined with the strategy, and can be reused in multiple transformations.

Support for strategies is provided by a number of transformation systems in various forms. In TAMPR [BHW97] the transformation process is divided into a series of rewrite rule sets. A sequence operator between the sets applies these sets successively. ELAN [BCD+00] provides non-deterministic sequential strategies, but lacks the ability to combine strategies with term traversal. Stratego [Vis01b] provides generic primitive traversal operators that can be used to compose generic tree traversal schemas. In the remainder of this section we therefore limit ourselves to a discussion of stratego.

A rewrite strategy is a program that transforms terms or fails at doing so. In the case of success, the result is a transformed term. In the case of failure, there is no result.

By combining the strategies with traversals we can control how and when rewrite rules are executed. Consider for example the topdown strategy:

```
topdown(s) = rec x(s; all(x))
```
[1]

[1]The recursive closure `rec x(s)` of a strategy `s` attempts to apply to the subject term the strategy obtained by replacing each occurrence of the variable `x` in s by the strategy `rec x(s)`.

The strategy expression `rec x(s; all(x))` specifies that the parameter transformation `s` is first applied to the root of the current subject term. If that succeeds, the strategy is applied recursively to all direct subterms of the term, and, thereby, to all of its subterms. This definition of topdown captures the generic notion of a pre-order traversal over a term.

Strategies *statically describe the execution of rewrite rules*. Rewrite rules are still *polluted* in case where dynamic information of the source program must be taken into account. Due to the limited source scope of rewrite rules only a *limited amount of context information* into account. So to ensure a correct scheduling, insight is required in all rewrite rules and in all strategies.

Local-to-Global Transformations The target scope of rewrite rules is limited to its source scope. This clashes with the fact that local-to-global transformations produce several subtrees which need to integrated in various locations in the target program. However, recall that these nonlocal subtrees can only be correctly injected into the target program when we can compute where they belong. In other words, such a rewrite rule should delay that injection until "the time is right". The concept of *scoped dynamic rewrite rules* is an extension of rewriting strategies that overcomes the limited target scopes of rewrite rules.

A dynamic rule is a normal rewrite rule that is generated at run-time and that can access information from its generation context. For example, to define an inliner, a rule that inlines function calls for a specific function can be generated at the point where the function is declared, and used at call sites of the function. *Dynamic rules can be used to implement local-to-global transformations* and basically allow you to *produce multiple nonlocal results* locally and delay their execution.

The execution of dynamic rewrite rules must be carefully managed. This is where programmable rewriting strategies enter the scene. Dynamic rewrite rules can be applied in arbitrary locations (under control of the rewriting strategy).

Nonlocal subtrees produced by local-to-global transformations can be integrated across the entire target program. In other words, arbitrary locations in the target program must be reached which *requires more complicated scope definitions and complex logic to control the execution of dynamic rewrite rules*. The basic scoping construct scopes dynamic rewrite rules according to particular subtrees. More complex scoping rules, than mere subtrees, are needed. However, it is unclear how more complex scoping rules can be implemented.

Controlling the execution of dynamic rewrite rules can either be done via rewriting strategies or by encoding additional logic in their left hand side. The former is generally preferred over the latter for reasons of clarity, programming reward, reuse (cfr. Section 3.1.4) and to avoid overly scoped and coarse-grained modules (cfr. Section 3.1.3). However, in case the logic to control the execution of dynamic rewrite rules is dependent on the context-information of dynamic rewrite

rules, then *the control logic becomes part of dynamic rewrite rules.*

Strengths and Weaknesses

+ Strategies explicitly support local-to-global transformations by allowing rewrite rules to exercise effects outside its scope and as such avoid dependencies caused by interacting traversals and rewrite rules.

− Strategies provide no features and constructs to integrate or modify without invalidating previously produced terms.

− Strategies statically describe the execution of rewrite rules, and a such statically define the places where local-to-global effects are executed.

3.2 Graph Rewrite Rules

Graph rewrite rules (see Figure 3.7) such as Fujaba [FNTZ98], Progress [Sch91] and AGG [ERT99] bear a lot of similarities to tree-based rewrite rule systems. The shift in representation, and their strong mathematical background creates a lot of opportunities that enrich the paradigm. Programs are stored in changeable data structures, which can be freely changed and extended by any rule. Furthermore, rewrite rules have large source and target scopes to support the implementation of global-to-local transformations and local-to-global transformations.

In this section, only the differences with respect to the tree rewrite rule paradigm are explained. Note that this discussion slightly deviates from the outline in Section 2.3 to improve readability. An in-depth overview of graph rewrite rules can be found in [BFG95].

3.2.1 Data Structures

Structure and Organization Unlike traditional rewrite rules, graph rewrite rules operate on "open ended" record structures or containers that represent the abstract syntax nodes. There is no predefined set of values that these containers may hold.

The topology of the edges (connections) between the nodes is a graph. In the context of MDA, where these systems are frequently used, the graph that contains the AST nodes of a program is referred to as the model.

There is no restriction on the number of nodes that are built and they do not need to be connected in a single graph. In other words, multiple subgraphs may coexist during the transformation process. Naturally, there is at least one (sub)graph representing the source program available to start the transformation process. This (sub)graph can be gradually rewritten as was the case with tree-base graph rewrite rules. Alternatively, a new (sub)graph can be created which

represents the target program. Auxiliary subgraphs or graph extensions are often used to implement complex transformations, see Section 3.2.2.

Construction Records are less rigid data structures than terms. One of the interesting possibilities is that upon creation, not all its neighbours need to be known. In other words, the node being constructed can be incomplete. The ability to *create incomplete nodes increases the separation between rewrite rules* as rules can restrict themselves to connections they need without anticipating, retrieving and handling all the neighbouring nodes.

Modification Containers are different from terms in that they can be easily modified without being recreated. This seems only a minor difference, but the *separation among rewrite rules is improved* as a rule can attach additional information, and its logic only needs to consider the information that is relevant. This flexibility also has its price, as it renders the data structures more *vulnerable to inconsistencies upon multiple rewrites*. A tree rewrite rule must consider all the parts and thus forces the developer to take these parts into account upon rewriting; in contrast a graph rewrite rule can ignore a part of the data structure upon a change possibly cause inconsistencies.

3.2.2 Transformation Modules

Similar to tree-based rewrite rules, a transformation module in graph rewrite rule systems is a graph rewrite rule, and transformation modules or groups of modules are not organized according the structure of the source language defined by its grammar.

Identification In graph rewrite rule systems, left hand sides are subgraphs which are matched against the graph under transformation. As subgraphs can arbitrarily include any set of connected nodes, a graph rewrite rule can also ignore any edge or node. This flexibility facilitates incremental development and growth of the graph, as each graph rewrite rule is only concerned with those edges and nodes it really depends upon. However, being totally ignorant about certain parts of a graph structure also endangers its consistency and correctness.

Subgraphs play a crucial role in the retrieval of context information. Like the tree rewrite rules, the *structure or topology of subgraphs*, when described as a collection of connected nodes, *is statically determined*. In the local-to-global paragraph, more sophisticated mechanisms are discussed.

Scope Similar to tree rewrite rules, the target scope of graph rewrite rules is also determined by the source scope. But in contrast to rewrite rules, the source

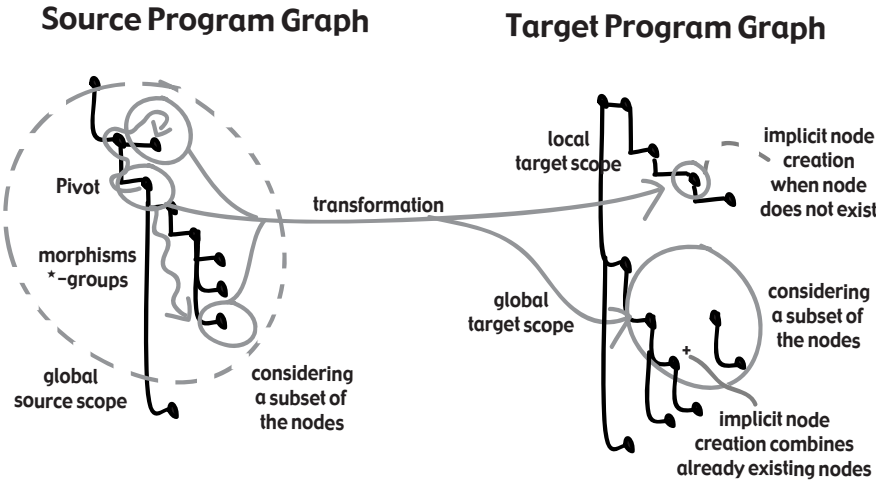

Figure 3.7: A schematic overview of a graph rewrite rule

scope is not limited to a mere subtree as any edge can be included in the subgraph of the LHS of a rule.

By including child-parent edges in the LHS of a graph rewrite rule, it can limit its applicability and as such control its scope. By including an edge and a node in the subgraph that denotes, for example, the parent, container or super component, the context information contained in those nodes can be used to scope rules. Combining this with the ability to partially include a node in the LHS, we get a specification which *fewer dependencies on the source graph* as compared to tree rewrite rules, a graph rewrite rule only has to include those nodes and edges that are relevant to narrow its scope.

Scopes can also be broadened, by leaving nodes or edges out of the scope definition.

Scheduling The scheduling of graph rewrite rules is largely the same as the scheduling of rewrite rules. We refer to Section 3.1.2 for a detailed discussion. The difference between the two formalisms emanates from the ability to produce multiple (sub)graphs. *Scheduling is less complicated* when the source program is not destroyed during the transformation process. At any time during the transformation process, we can consult that structure. However, the ability to change that structure and add information complicates matters. Rewrite rules *implicitly cooperate as they have to agree on the same underlying graph structure* (source, target, run-time information) as a prerequisite for a consistent implementation.

Effect *Graph rewrite rules explicitly support transformations with global source and global target scopes. This is discussed in more detail in the subsections local-to-global and global-to-local.*

The effect of a graph rewrite rule is stated in its RHS. However, when only taking the RHS into account, one cannot distinguish the nodes created at the RHS. By comparing its RHS with its LHS, one can deduce which nodes are to be created. The nodes that are not matched by its LHS, are new nodes. As its LHS can also match nodes which are created by other rules, *graph rewrite rules implicitly depend on other rules.*

In a graph rewrite rule implementation, it is often impossible to determine which rule will be triggered in advance. Hence, it is also in general impossible for transformation architects to determine which rule should explicitly create the new node. With the ability to create partial nodes, one can shift from explicit to *implicit node creation.* Implicit node creation creates a node when necessary. The node in the RHS is first matched against already existing nodes. If no match has been found, the node is created. This allows for a *more fine-grained modularization* of graph rewrite rules in so-called aspect-driven transformations [DGL+03]. This mechanism *avoids that graph rewrite rules are get polluted with scheduling, and the semantics of other rules.*

When nodes are matched either by the left hand side of a graph rewrite rule or through the implicit creation of the right hand part, the partial definition needs to be taken into account: rewrite rules only needs to consider those parts that are relevant and can remain oblivious to the rest of the definition. Of course, *the same hazards of data corruption remain present.*

Dealing with a completely defined node is easier than with a node that can be arbitrarily incomplete. In the former case, it is easier to anticipate the neighbours that can be found and how the modifications will affect the whole node. Moreover, when nodes are partially complete, they sometimes temporarily represent incorrect code fragments of the target language. The correct procedure to complete them and render a semantically and syntactically correct code fragment demands a custom strategy codetermined by the code fragment. Unfortunately, there is a lack explicit integration semantics in graph rewrite rules (see next paragraph).

Local-to-Global *Broader source and target scopes* of graph rewrite rules *ease the definition of local-to-global transformations.* The LHS can match distant nodes in the target graph and manipulate the target program. *Local-to-global transformations remain dependent* on the:

- the results produced by *other* transformations and their *interactions*;
- the *exact properties* of the graph at a *certain state* of the transformation process;
- a *particular scheduling*;

- the *semantics* of the results produced by other transformations erected by the *interwoven compositions* that encode the translational semantics. There is thus a lack of explicit integration semantics.

Note that careful use of rewrite rules by a strict design scheme can reduce some of these dependencies see Section 3.1.2 and Section 3.1.3.

Global-to-Local Graph rewrite rules have large source scopes which *facilitates the retrieval of context information.* More precisely, left hand sides are subgraphs which are matched against the graph under transformation. Such LHS subgraphs allow the acquisition of any additional context information while transforming a particular node. By including an edge and a node in a LHS subgraph (e.g. the parent container or super component) the context information contained in those nodes can be used to construct a RHS subgraph.

In order to access an ancestor of a node, one must include every intermediate node along the path to reach that node. *The subgraph matched by a rule is thus highly dependent on the graph structure, which cripples its maintainability and reusability.* However, besides the engineering quality downsides in the case the selection of a neighbour depends on the content of the graph, it is very cumbersome, if not impossible to statically predetermine the path to the node as there are, in the worst case, an infinite number of possibilities.

For the above reason, rules are extended to cope with certain variations in the matched subgraph structure. The description of a subgraph is augmented with annotations attached to individual edges and nodes. These edges and nodes denote possible matching variations. As such, graph rewrite rules are able to abstract over various static (fixed) paths. In general, the annotated edges or nodes are isomorphic with a set of those edges and nodes. A popular example of such annotations for edges are $*$. Edges annotated with a $*$ coincide with a chain of an arbitrary number of edges [HE92]. The $*$ (of Δ-rewriting [KLG91]), denote zero or more occurrences of annotated graph elements. These extensions allow us to cope with more flexible paths. However, complicated logic cannot be captured and needs to be handled by other rewrite rules. *As such, rewrite rules that compute context information implicitly cooperate with the rewrite rules that consume this information.*

In general, we can state that *subgraph matching* is accomplished by computing a morphism between a subgraph and the graph currently under transformation. As shown in the previous paragraph, subgraph matching is both a blessing and a curse. The use of these general morphisms means that rewrite rules easily give rise to unexpected matches. *In order to compensate for these mismatches, the definition of rewrite rules must be changed.*

Although source scopes are larger, compared to tree rewrite rules, it is not always easier to obtain context information. Every piece of information required and the location of that information, needs to be described by the LHS. The

matching abilities in the LHS of graph rewrite rules are less powerful compared to the traversals and generic tree traversal operators provided in extended rewrite rule formalisms. Therefore, the retrieval of complex information is separated often out of the rule requesting the information and must be provided by other rules. Unlike typed tree rewrite rules where additional information requires the rewriting of the term with a suitable arity, graph rewrite rules can simple add the necessary information. Hence, *there are less dependencies between graph rewrite rules that add information to the same node*, compared to tree rewrite rules. The computation of the transitive closure for class inheritance illustrates this concept. However, to determine what nodes and edges must be added to provide the necessary context information, the graph topology of the source program must be compared to the LHS of the rule. In other words the context information is implicit. As there is very little or no control over the execution of rules and as rules can change information, *it is difficult to tell if the computed information is in a valid state before other rules are applied.*

Strengths and Weaknesses

\+ Graph rewrite rules support implicit global-to-local transformations through flexible data structures which can be incrementally changed and updated.

\+ Graph rewrite rules have large source and target scopes to support the implementation of both global-to-local and local-to-global transformations.

\+ Graph rewrite rules with implicit node creation encode a symmetric model. Local-to-global transformations merely produce multiple results. The fact that some of these multiple results affect the results produced by other (kinds of) transformations, is taken care of by the graph rewrite system.

\+ Graph rewrite rules with morphisms facilitate the identification of local-to-global transformations.

− Graph rewrite rules suffer from dependencies on intermediate graphs.

− Graph rewrite rules lack explicit integration semantics.

− Graph rewrite rules can change every term which may invalidate previously computed information.

3.3 Macros

A macro (see Figure 3.8) can be understood as a function that accepts a parameterized code fragment as an argument and generates a new code fragment as a replacement for the original code fragment. Compared to other approaches,

the macro facility of a host language is a compile-time embedded rewrite rule system using concrete syntax[2]. The major difference between macros and other approaches described in this chapter is that only a part of a source program is subject to rewriting, and that each macro contains its syntactical definition and its translational semantics.

We focus on macro systems developed for Lisp dialects (i.e. Common Lisp and Scheme) because most important advances in macro systems have been and are still being made in those languages.

A standard example for a Lisp macro is `with-open-file` (see below). Its purpose is to hide the program details ensuring that a file is opened so that it can be processed within a block of code, and more importantly that it is finally closed on return from that block. The `unwind-protect` function (which is the Lisp counterpart of Java's `try-catch`) ensures that the file is closed in case an exception is thrown.

```
(defmacro with-open-file ((var filename) &rest body)
  `(let (,var)
     (unwind-protect
         (progn
           (setq ,var (open ,filename))
           ,@body)
       (close ,v))))

    `(let (,var)
```

3.3.1 Data Structures

Structure and Organization Lisp and Scheme typically represent a program with an abstract syntax tree. Besides the program itself, there are additional data structures. As macros are expanded at compile time, some compile-time data structures are made accessible to macros.

Construction In Lisp, there are various ways to construct code because code is merely a nested list.

Modification There is no equivalent to the *modification* and *deletion* operators that occurs in other approaches because macros are by nature applicative: they map program fragments to program fragments without any side effect to an abstract syntax tree.

[2]This comparison even holds for Lisp. In Lisp, code fragments are represented as lists. However, these lists are also the syntax used for writing programs as a developer.

3.3.2 Transformation Modules

A transformation module is a macro. As a macro combines a syntax description and a semantical definition, a language implementation is organized according the structure of the source language.

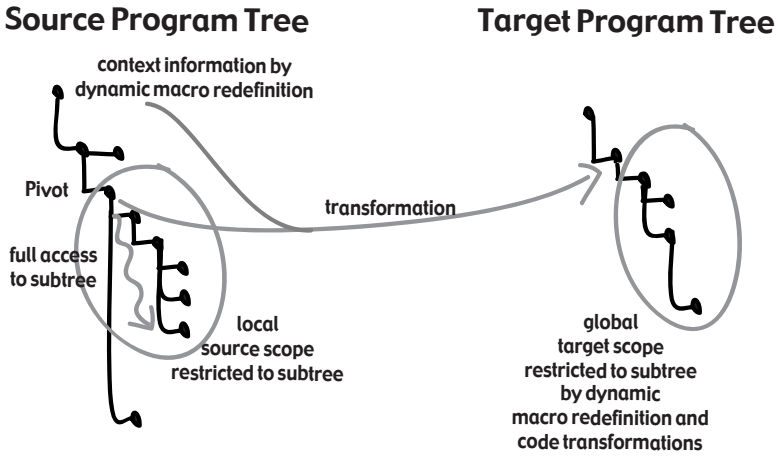

Figure 3.8: A schematic overview of a macro

Identification As macros operate at compile-time, the expression is treated as a code fragment. The code fragment that is passed to a macro for expansion is *destructured*. This is done implicitly in the parameter declarations of macros.

Scheduling Macros are scheduled automatically by the macro expander which is built into the compiler of the host language. Unlike rewrite rules, macros do not have a conditional clause in which restrictions on their applicability can be described or in which the invocation of other macros can be explicitly triggered. *Macros which match the context needed by other macros are polluted* with the additional responsibility to define or provide the latter macros (see global-to-local).

Scope The source scope of a macro is bound to a single expression. The primary source of information comprises the program fragments which are bound to variables in the destructuring step. These can be further accessed by list operations. As these program fragments may contain other macros, *macros are dependent on one another.*

The compile-time data structures are made accessible to macros by extending the local scope of macros. More details follow in the global-to-local paragraph.

Effect The resulting code produced by a macro must be *constructed* from the parts determined by the code destructuring step.

As already mentioned, a macro is passed a list representation of a code fragment in order to produce another code fragment as a replacement. In principle, list processing operations are sufficient to analyze and generate the code fragments. Moreover, one implication of the fact that macros are functions that transform code fragments is that the full (Turing-complete) language of the respective Lisp dialect can be used for defining macros.

Global-to-Local There are two directions from which a macro can obtain context information, either from the surrounding code or from the embedded code.

The most straightforward way of using information from the surrounding code is to plainly use the names of the variables defined in the lexical scope (and also dynamic scope in the case of Lisp). This is called *intentional variable capturing. The implicit dependencies between macros render the program that use them error prone and hard to debug.*

Due to the embedded nature of macros, information of the surrounding code can be obtained by accessing the surrounding lexical environment of the macro invocation. *The result is again a macro definition containing implicit dependencies*

The problem to access information in the surrounding code is that the data structure of macros is limited; macros cannot ascent from their parameters to retrieve information from the surrounding code. Passing information from one macro site to another macro site must be done explicitly by locally redefining macros. This mechanism is similar to the dynamic rewrite rules of stratego (see Section 3.1.4), but much weaker, as information can only be passed from a broader scope to a smaller one (i.e. scope defined by the expansion process). So whenever a macro requires context information, *another macro (higher up in the program) must be changed to redefine the macro.*

Context information from the parts of a macro can be obtained by inspecting the destructured code. Code is merely a list can thus be accessed with any list processing function. However, traversal logic is not natively supported and must be implemented with a library. As macros are resolved by a top-down traversal and are expanded along that traversal, *macros are able to access other macros and change them.* In that regard, macros are as problematic as rewrite rules, with the addition that *a macro can contain embedded implicit references to other macros.*

In order to access information from arbitrary locations, there is even an additional challenge. The challenge stems from the fact that not all source program expressions are subject to expansion. In order to implement such global-to-local transformations, a common ancestor is required for the source information and for the target where context information is requested. At that common ancestor,

we can query for information and redefine the macro to inject that requested information. If the common ancestor is not a macro, we have to create a new macro. In case there already exists a macro, *the definition of the existing macro has to change.*

Local-to-Global *The implementation of local-to-global transformations in macros is not possible*, as only a single result can be produced. Moreover, a valid code fragment must be produced, prohibiting the use of intermediate programs with artificial constructs to contain both local and nonlocal code fragments. Technically speaking, a macro could emit another macro, say `NL`, artificially combining local and nonlocal results. However, due to lack of control over the expansion process, one cannot prohibit the full expansion. Consequently, the `NL` macro would not always remain available. Certain cases of local-to-global transformations can be facilitated by using local macro redeclaration: a local macro can transport nonlocal results of a broader scope to a narrower scope.

Strengths and Weaknesses

− Macros can implement global-to-local transformations as long as the required information resides in the destructured variables. Otherwise, other macros which provide the necessary information embed implicit references.

\+ Macros can access compile-time information because they are well-integrated in the compiler.

\+ Macros support local-to-global transformations within the lexical scope through dynamic creation of other macros. In general local-to-global transformation cannot be implemented.

− Macros lack explicit integration semantics (see Section 3.1.2 on page 51, and Section 3.2.2).

3.4 Template-based Approaches

Template-based transformation (see Figure 3.10) systems such as XSLT [Cla99], Frames [SZJ02], LMP [Dev98] and Velocity [Vel03] are a very heterogeneous group, as many of these systems are based on various technologies and paradigms. However, they can be treated as a single group for the purpose of this discussion, as we do not focus on technical implementation details.

One of the most characteristic features of template-based approaches is that the transformation process is driven by the target language or target program. This means that the focus of modularization is defined in terms of the target

program. More precisely, a transformation (or template) consists only of a right-hand side that produces a program fragment. The transformation has the entire source at its disposal to execute its right-hand side. In most approaches, they can be combined and are able to cooperate by passing information from one transformation to another.

Template-based languages are known for their powerful and robust query mechanisms to implement global-to-local transformations. It is a common belief that template-base approaches are better suited to implement local-to-global transformations [vWV03]. We argue against that statement in Section 3.4.2.

The example shown in Figure 3.9 ranks authors by the total number of books they sold, sorted in decreasing order. The XSLT fragment contains two templates. The first template operates on the entire source and retrieves the authors (`...select="/publisher/authors/author"`). These are then subsequently transformed. The call to `apply-templates` invokes the second template. The select attribute acts as a parameter for the second template that further processes the author. The result of this template is put inside a `bestsellers-list` tag.

3.4.1 Data Structures

Structure and Organization Each technique uses a different structure to represent programs, hence there are hardly any commonalities on the representation level. Like most transformation approaches, terms represent the abstract syntax of a program and are organized in a tree. The tree is the result of composing the structures with other structures. In some approaches [Vel03, Cla99], the structure and organization is even completely hidden. This is achieved by embedding transformations in the target programs which need to be produced.

Creation As the transformation process is organized according to the structure of the target program, each transformation produces a complete fragment of the target program. All the information required to construct the fragment must thus be derived from the source representation and must also be known upfront. The impact of this becomes clear when dealing with local-to-global transformations.

Modification Modification of the already produced target program fragments is, although technically realizable, very uncommon. The data structures are read-only. Once produced, the structure can no longer be changed. Changeable data-structures would clutter up the strict target-driven architecture of the process. *Moreover, modification can even be considered as an attempt to break the encapsulation of transformation modules, because there is no mechanism to control the modification and ensure the functionality of the target program fragment [Bri05].*

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<bestsellers-list>
  <xsl:apply-templates select="/publisher/authors/author">
    <xsl:sort select="sum(/publisher/books/book
              [author- ref/@ref=current()/@id]/sold)"/>
    <xsl:sort select="last_name"/>
  </xsl:apply-templates>
</bestsellers-list>
</xsl:template>

<xsl:template match="author">
  <copy>
    <name><xsl:value-of select="last_name"/>,
    <xsl:value-of select="first_name"/></name>
    <total_publications>
      <xsl:value-of select="count(/publisher/books/book[author-
         ref/@ref=current()/@id])"/>
    </total_publications>
    <total_sold>
      <xsl:value-of select="sum(/publisher/books/book[author-
         ref/@ref=current()/@id]/sold)"/>
    </total_sold>
    <rank><xsl:value-of select="position()"/></rank>
  </copy>
</xsl:template>

</xsl:stylesheet>
```

Figure 3.9: XSLT transformation to rank authors by the total number of copies of books sold in decreasing order.

3.4.2 Transformation Modules

A transformation module is a template. Templates organize a language implementation according the structure of the target program.

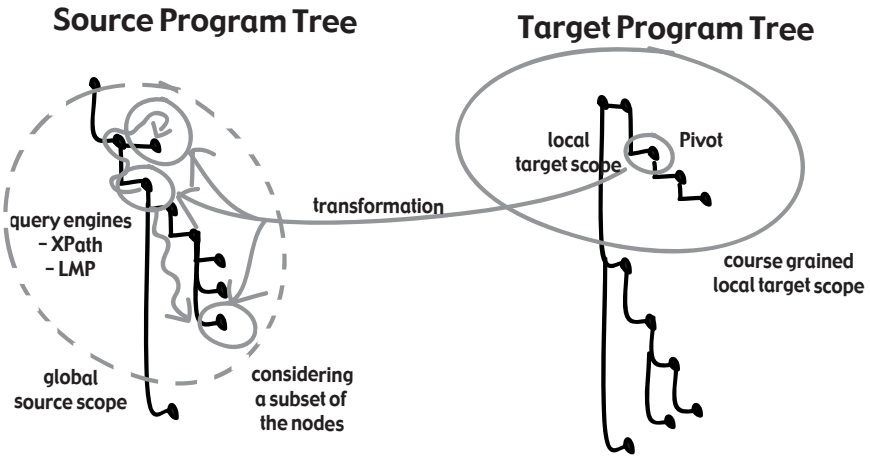

Figure 3.10: A schematic overview of a template

Identification Identification of the necessary parts of a source program is supported by a query engine. As Van Wijngaarden [vWV03] states, the availability of a sufficiently powerful query mechanism is an important prerequisite for target-driven transformations. Such a mechanism makes it possible *to easily look up information anywhere in the source*, based on some conditional expression. In LMP, the underlying logic engine is used to query the source program. XSLT uses a combination of XPath and XQuery. *Templates which contai queries are dependent on the whole source language.*

Scheduling In pure template-based approaches, the scheduling of templates is completely handcrafted and embedded in them. This means that templates explicitly invoke one another and combine the resulting programming fragments into a larger one. *These explicit invocations result in a very rigid and tangled language implementation.* However, modern template-based approaches are a mixture of source-driven and target-driven. This means that the source program is used to invoke the templates which are applicable. In XSLT for example, the phrase `<xsl:apply-templates select="/publisher/authors/author">` invokes all templates which can produce a code fragment for an author.

Scope The source scope of a template is always an entire source language. Templates may be parameterized by a source program node. For example, the second XSLT template `<xsl:template match="author">` is parameterized by the `author` source program node. However, this mechanism merely parameterizes source scopes, but does not restrict them as query languages in template engines enable every template to access the entire source program.

The target scope is always local. *It is up to the caller of a template to properly and explicitly consume all its produced results.*

Effect The resulting code can be constructed by basically using the entire source program and by invoking explicitly or implicitly other templates. The target code is usually constructed through composition. As change cannot be made once a structure is produced, the target program structure must be complete and correct (see modification).

Global-to-Local The retrieval of information plays a crucial role in template approaches because target program fragments must be complete upon construction. The query mechanisms in XSLT such as XPath and XQuery are designed to locate any node in the tree using a *structure-shy path*.

The background of structure-shy paths lies in structure-shy or adaptive programming (AP) [Lie96]. It was introduced to more easily adhere to the law of demeter [LH89]. In short, the law of demeter restricts the number of acquaintances of an object by prohibiting invocations of methods of a member object returned by another method. The naive way of adhering to this principle is scattering the logic in the application's object graph. In response to the maintenance problems introduced by this scattering, AP introduced the notion of structure-shy behavior which allows visiting the objects contained within an application locally without explicitly describing the structural relationships among these objects. Structure-shy behavior is described by *structure-shy traversals*. As they describe an abstract path through the object structure, hence the name *structure-shy paths*. Although AP is described in an object-oriented setting, it can also be applied on other data structures such as XML.

A structure-shy path does thus not detail the actual path to be followed. Consequently, *such paths reduce the dependencies between transformation modules and the source language* rendering them more robust to changes than fully specified paths. Despite the powerful query mechanisms, global-to-local transformations challenge a fine-grained modular design. Since a transformation has the entire source program at its disposal to query for information, transformations are often *coupled with the source language*.

Local-to-Gobal As templates are modularized according to their target program fragment instead of the translational semantics of a source program node,

transformations that scatter code in the target program become a global-to-local problem. As global-to-local problems can be handled quite well, templates do not seem to have the additional problem of handling local-to-global transformations compared to source-driven transformations. In fact the contrary is true. In Section 2.3.2, we showed that the direction of a transformation does not change the impact of scope on the modularization of transformations. This position is supported by Brichau in [Bri05] which introduces a technique to support local-to-global transformations with integrative composition on top of the template-based approach LMP.

Local-to-global transformations are a complex problem which can only be tackled if all aspects are taken into account. Figure 3.11 schematically illustrates the challenges local-to-global transformation pose in template-based approaches. The figure depicts uses relationships which denote the information that is necessary to construct target program fragments. The following four issues show that *the modularization of local-to-global transformations is broken due to scattered management*:

1. First, both source or target driven approaches need to transport nonlocals to the correct locations. Templates query the source program to construct nonlocals or query the templates to obtain nonlocals. So technically speaking there is not much difference with source driven approaches. Furthermore, when nonlocals possibly can become part of different targets, then *the logic of the local-to-global transformation is duplicated (use_2 in t_3 and in t_1) and scattered (use_3 and use_2 in t_3) over the different targets.*

2. Second, the integration of nonlocals which stem from various translational semantics has become the responsibility of a single target language constructor. The result is a super-transformation which knows how to compose nonlocals produced by other transformations. *These super-transformations are tightly coupled and highly dependent on other transformations.*

3. Third, nonlocals that need to become a particular part of another target fragment must be easily identifiable. Note that only a small amount of program fragments bear names. Moreover those names are not fully qualified as they are qualified through their source contexts. Hence, in these cases, *qualification introduces strong dependencies with other transformations.*

4. Fourth, the integration of nonlocals may depend on the source program or on the target program context. Source program context dependencies are embedded in templates and must be managed explicitly. Target program context dependencies are even harder to implement as the target structure is hidden (cfr. Section 3.4.1). *These dependencies must be explicitly resolved by every template, resulting in scattered management.*

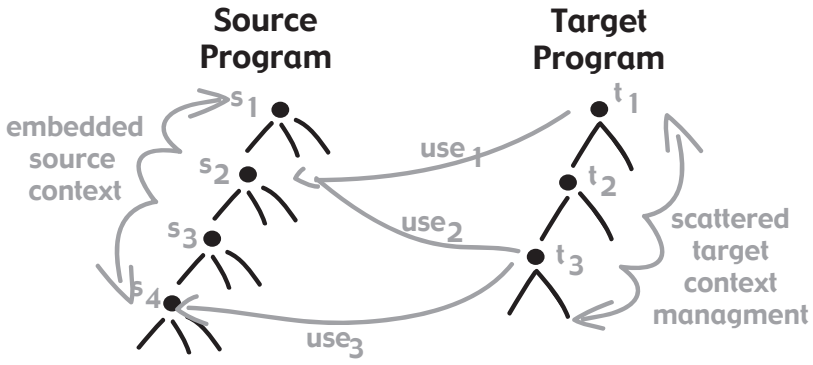

Figure 3.11: A schematic overview of some of the challenges when implementing local-to-global transformations by template-based LDTs. The uses relationships denotes the information that is necessary to construct target program fragments.

Strengths and Weaknesses

– The integration logic of local-to-global transformations is duplicated and scattered over the various templates that produce the targets in which the nonlocals must be integrated.

– Embedded global access to source trees increases the dependencies among transformation modules.

\+ Structure-shy paths/queries are robust mechanisms to locate information.

– Templates cannot access the produced target program and thus cannot change it once it is produced.

– Templates do not offer explicit integration mechanisms.

3.5 Attribute Grammars

An attribute grammar [Knu68, Paa95] is a context-free grammar augmented with attributes, semantic rules and conditions. For each syntactic category X of N in a grammar (see Definition 4.1), there are two finite disjoint sets of inherited and synthesized attributes. Each attribute takes a value from some semantic domain (such as the integers, strings of characters, or structures of some type) associated with that attribute. An attribute is defined (see Figure 3.15) by semantic functions or semantic rules associated with productions.

Attribute grammars are known for their modular implementation of local-to-global transformations by means of implicit copy rules, explicit copy rules and templates.

The basic program transformations are semantic functions or semantic rules. The functions can access the values of attributes from their parent (inherited attributes) and the values of attributes from their children (synthesized attributes). In semantic functions, attributes are retrieved by value or by reference [Hed99] and computation is expressed in an declarative style.

In Figure 3.12, a simple document definition language is presented that consists of sections (`Section`) and text fragments (`Pcdata`). Its representation in HTML is defined by the `Doc` attribute, which in case of a section prints its title in bold and appends its body, and in case of a text fragment prints its contents as a paragraph.

```
data Doc
    | Section title : String body : Docs
    | Pcdata string : String

Sem Doc
    | Section lhs . html = "<bf>" ++ title ++ "</bf>" ++ @body.html
    | Pcdata lhs . html = "<bf>" ++ string ++ "</P>"
```

Figure 3.12: Simple document definition language defined with an attribute grammar (upper part). The `Doc` attribute (lower part) returns the HTML code.

3.5.1 Data Structures

Structure and Organization Attribute grammars operate on named terms or read-only record structures that represent particular nodes of the abstract syntax tree of the source program. As in other approaches, a term is defined through the composition of a set of other terms, hence setting up a tree structure.

The source program tree and target program tree are immutable. The target program is stored and constructed in a different data structure. The target program data structure is not the only data structure that can be used during the transformation process. Any other can be used and constructed when needed.

Construction An attribute is typed with a particular domain that denotes its set of possible values. Plain attribute grammars only allow attributes to be typed with a primitive domain, for example integers, floats, strings, etc. Those domains appear to be too basic and in the case of strings too unstructured to represent the target programs, but according to the basic attribute grammar school, the primitive domains suffice. The reason for this is discussed in the local-to-global paragraph in more detail.

With the advent of higher order attribute grammars [VSK89], this “limitation” is removed by unifying the domain of parse trees and the domain of attributes.

In other words, the domain of an attribute can be a grammar of (another) language. This is a particularly important feature in multi-phased compilers where subsequent phases can continue computing on the results of the previous phase.

The grammar domain of attributes is represented with terms, just as the source abstract syntax tree. The term needs to be complete [Hed92], so upon creation, all the subterms must be computed in advance. The impact of this becomes clear when dealing with local-to-global transformations.

Modification Attributes are computed and information that is necessary for that computation is computed first. Hence, there is no need for an ability to modify the produced result afterwards. Therefore, the primitive attribute domains such as strings and string concatenation are initially sufficient to represent and construct programs. The absence of modification enables us to firmly rely on the value of an attribute in other attribute evaluations. This simplifies the reasoning of a language implementation and reduces scheduling complexities.

3.5.2 Transformation Modules

Modules in attribute grammars formalisms depend on the specification languages used to describe them. Hedin pioneered in modular attribute grammar research by designing object-oriented attribute grammars [Hed89], where modules are classes which align with the syntactic categories of source language grammars. With [Hed92], values (or objects) can be accessed by reference. Reference semantics enables flexible attribute reevaluation in incremental semantic analysis and facilitates the implementation of static analysis. She, Mernik [MuLA99] et.al. and Grosch [Gro92] continued investigating the use of object-oriented techniques like inheritance and multiple inheritance. Recently, she pursues more advanced object-oriented concepts like aspect-orientation in order to further modularize the implementation of languages in attribute grammars. Modules in attribute grammar systems like JastAdd [HM03] but also in functional languages [Swi, OdMS00] no longer necessarily align with the syntactic categories of the grammars. In this setting, a module is defined as an aspect of the semantic behavior of a syntactic category, e.g. name analysis, type checking, (intermediate) code generation, etc. Each aspect contains several attribute definitions for several syntactic categories. Hence, an aspect crosscuts the definition of a syntactic category.

Consider for example the grammar definition in JastAddII of a small expression language in Figure 3.13. The nodes in an AST are viewed as instances of Java classes `Expr`,`Add` and `Id`. These classes are arranged in a subtype hierarchy using the `extends` keyword: the `Add` and `Id` are subclasses of the `Expr` class. An AST class is introduced with the `ast` keyword and corresponds to a nonterminal (in case of the `Expr` class) or a production (in case of the `Add` and `ID` class) and may define a number of children and their declared types, corresponding to a production's right-hand side. Aspects can be specified that define attributes,

equations, and ordinary Java methods of the AST classes. An example specifying a very simple type-checking algorithm as an aspect is shown in Figure 3.14.

```
// Expr corresponds to a nonterminal
ast Expr ;
// Add corresponds to an Expr production
ast Add extends Expr : : = Expr leftOp, Expr rightOp ;
// Id corresponds to an Expr production, id is a token
ast Id extends Expr : : = <String id> ;
```

Figure 3.13: An example grammar definition in JastAddII

```
// Declaration of an inherited attribute env of Expr nodes
inh Env Expr.env() ;
// Declaration of a synthesized attribute type of Expr
// nodes and its default equation
syn Type Expr.type()=TypeSystem.UNKNOWN ;
// Overriding the default equation for Add nodes
eq Add.type() = TypeSystem.INT;
// Overriding the default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

Figure 3.14: Type checking aspect in JastAddII

Identification In plain attribute grammars, attributes are globally declared and typed for the whole abstract syntax tree. Individual attribute computations or definitions are attached to the abstract syntax tree nodes of source programs. This way, attribute definitions could refer to attributes without having to stipulate where the attributes are computed. In modularized attribute grammars, attribute definitions are declared per nonterminal. Although for example inherited attributes are then declared locally without a definition, these local declarations implicitly refers to an actual attribute which is declared and defined in other non-terminals.

Scope The source and target scopes of an attribute computation are strictly restricted to the abstract syntax node it is attached to. *These small source scopes render an attribute computation highly independent on the rest of the language. The only dependency that remains is the choice that has to be made between either an inherited or a synthesized attribute depending on the whereabouts of the information.* If contextual information is needed, then an inherited attribute is

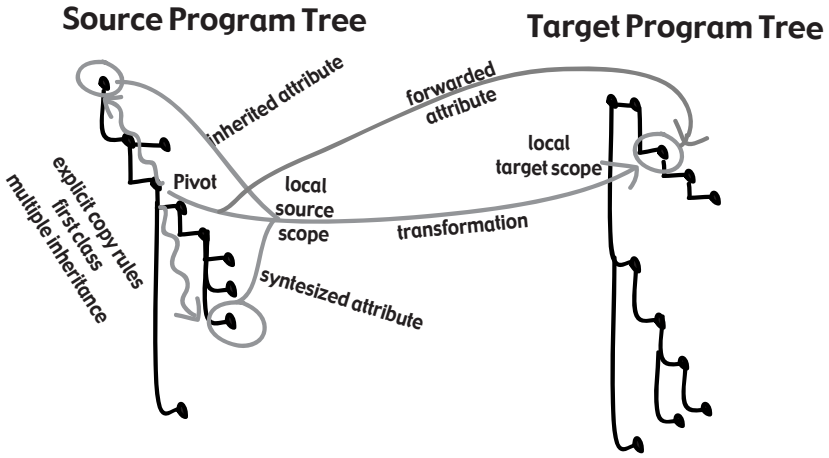

Figure 3.15: A schematic overview of an attribute definition

used. This erects a dependency because one assumes that the context is providing this information. If other nonterminals in the context of a given nonterminal need this information then a synthesized attribute for that nonterminal is used. This erects a dependency because one defines a synthesized attribute to be used for the other nonterminals.

There are attribute grammar extensions/implementations that extend the small source scope such as (multiple) inheritance attribute grammars, forwarding and first class attribute grammars. The former (e.g. [Gro92, HM03]) allow, orthogonal to the inheritance of attribute definitions along the alternatives of a production in a context-free grammar specification, the inheritance of attribute definitions along other design dimensions e.g. semantic aspects [Paa95]. Inheritance is used for example to thread an environment attribute through an attribute grammar. With forwarding [WdMBK02], attributes can be redirected to their translational semantics. Another approach to extend the source scope is by turning the propagation of information via attributes into first class citizens. As such, a single propagation scheme applies for multiple productions. These approaches are discussed in more detail in the next subsections. *Even with these extensions, dependencies among attribute definitions do not increase as attribute definitions remain unaware of them. In other words these extension operate implicit.*

Scheduling The execution of a transformation described in attribute grammars is driven by the need of information. Whenever information is required during the execution of an attribute, the attribute containing that information gets computed first. Attribute grammar compilers deduce an order of execution.

Effect The computation of attributes is defined in terms of the information contained in the production and in terms of inherited and synthesized attributes. Hence, there is no need to 'directly' identify other parts of a source AST. It suffices to simply request the information which is necessary. It would actually go against the philosophy of the paradigm to directly identify particular nodes of an AST, as it *breaks the modularity of an attribute definition*. We emphasis on the word "directly", because evidently the computation of an attribute may yield another AST node. Note that this does not mean that direct references to other AST nodes are not useful. In reference attribute grammars [Hed92], attribute computations may lead to references to other AST nodes on which in turn attributes may be requested. This is used to simplify the implementation of semantical analyses.

Global-to-Local with plain attribute grammars The ability to access attributes from other parts of a source tree is crucial to be able to implement global-to-local transformations. By limiting the access to attributes of the parent term and of the subterms, *attribute definitions are modular as they do not explicitly communicate with other language constructs in the source language*.

Except in reference attribute grammars [Hed92], attribute computations are simple functions which cannot be straightforwardly parameterized to render the computation dependent on the "sender". An attribute computation could request another attribute that serves as a parameter which must be computed by the sender. Although this enables interesting possibilities to adjust the parameter depending on the context, the hazards of accidentally changing or capturing this parameter outweighs the gained advantages of parameterization.

In order to be able to communicate attribute values across larger tree fragments, all intermediate nodes must be explicitly aware of that value. This involves copy rules for each intermediate node to either copy the attribute up to its parent or down to its subterms. These implicit rules are produced by the attribute grammar, providing that they can be deduced. Implementing more complex source program queries than merely ascending or descending is a hassle (see queries to analyze source code [Kel07]).

Global-to-Local with explicit copy rules Attribute grammar systems provide implicit copy rules so as to copy inherited attributes from ancestors or synthesized attributes from descendants. Communicating values using implicit copy rules is limited. More complex computations are necessary for instance in the case of synthesized attributes. In response to that problem, schemes (with limited strengths) have been developed in attribute grammar implementations on top of the formal specifications to alleviate the developer of those tedious rules. Examples of such specifications can be found in First-Class Attribute Grammars [OdMS00] and in Single Inheritance Attribute grammars [HM03], (Multiple)

Inheritance Attribute Grammars [MuLA99].

First-Class Attribute Grammars introduce three new abstractions in an attribute grammar specification: families, rules and aspects. *Families* model sets of input or sets of output attributes. *Rules* map families of input attributes of a production to a family of output attributes. *Aspects*[3] map production names to rules. As such, common attribute copy rules can be reused.

Single Inheritance Attribute Grammars [Hed99] allow attribute definitions to be inherited along the alternatives of a production in a context-free grammar specification. Additional nonterminals can be defined which are not motivated by the context-free grammar, but from a reuse point of view. These nonterminals are called *semantic nonterminals*. Productions that define semantic nonterminals (i.e productions having such nonterminals on their RHS) are used to extend the behavior of other nonterminals with the attribute definitions defined in the semantic nonterminals.

Multiple Inheritance Attribute Grammars provide templates to express common attribute communication patterns. A template is a polymorphic abstraction of a semantic function parameterized with attribute occurrences which can be associated with many production rules with various nonterminal and terminal symbols.

All these approaches facilitate the implementation of local-to-global transformations by allowing information to be more easily *transported from one nonterminal to another nonterminal without having to involve all intermediate nonterminals. The attribute definition requesting the information does not has to change, as such this external mechanism respects the modularization of the existing attribute definitions.*

Global-to-Local with implicit copy rules Forwarding [WdMBK02] is a technique for providing default or implicit attribute definitions in attribute grammars by redirecting requests to the translational semantics of language constructs. The primary goal of this technique is to ease the extension of a language implementation with new modular language constructs. Extending a language with a new language construct in attribute grammars requires that the new construct provides the correct definitions for the attributes that are declared. As such, the new language construct participates correctly in the already existing language constructs.

As the semantics of new language constructs are defined in terms of its translation to an existing language construct, the redefinition of attribute definitions of new language constructs would require detailed knowledge of all semantic aspects of the base language. In order to avoid this, "copy rules" for all relevant attributes are automatically generated by forwarding. As such, *new language constructs can*

[3]Aspects in First-Class Attribute Grammars are not to be confused with the concept of aspects found in Aspect-oriented Programming [KLM+97]

respond to attributes provided by their translational semantics without needing to pollute the new language constructs with those definitions.

Forwarding is also different from typical object-oriented extensions of attribute grammars since the construct to forward to is dynamically computed at attribute evaluation time instead of statically determined via inheritance when the attribute grammar is defined.

Local-to-Global Nonlocal results produced by local-to-global transformations must be distributed over a target program. As the value of an attribute is immutable once computed, all the parts of a new target program fragment must be collected and combined, upon the creation of that target program fragment.

In case the necessary parts are not located in a subtree, *the retrieval logic is distributed over various attribute definitions.* Attribute computations must be altered also to retrieve and combine all the parts. *Hence, attribute definitions are no longer strictly concerned with translational semantics but are cluttered with other concerns.* The combination of various subtrees is a bit more problematic, as attributes are immutable. The attribute could be reconstructed, but that *requires detailed knowledge about the program fragment.*

Strengths and Weaknesses

− Attribute grammars force global-to-local transformations to be implemented like local-to-global transformations. Hence, attribute definitions are no longer strictly concerned with the translational semantics but are cluttered with other concerns.

\+ Attribute grammars modularize the implementation of local-to-global transformations with implicit copy rules, explicit copy rules and templates.

\+ Language extensions implicitly rely on attributes of the extended language by means of forwarding.

3.6 Compositional Generators

Although, compositional generators (see Figure 3.16) do not define a language, we do consider them in this dissertation because they provide interesting techniques to compose various software components and fragments. These compositions are important to implement local-to-global transformations. Furthermore, we will adopt some of their techniques to modularize the implementation of languages along their language constructs (see Section 6.3.3).

The approaches, discussed in the previous sections, are called transformational generators. They transform a program in one language to a program in another language. Programs written in a source language are transformed by *constructing*

for each expression of a source program the necessary new expressions written in a target language in order to build up a semantically equivalent target program.

Compositional generators do not construct individual expressions, they rather produce programs by composing several smaller programs or program fragments together. The selection and composition of a number of smaller programs can be considered as the source program, along with some additional configuration information. The set of composition rules, constraints and selection primitives form the source language in that respect.

Compositional generators advocate a symmetrical view on program fragments. Each fragment is treated as a separate concern of the overall program which can/need to be combined with other fragments. In other words, no distinction is made between a local-to-global or local-to-local transformation.

Throughout this section, three techniques are used to illustrate compositional generators.

GenVoca [BST+94] is a technique to produce software through the composition of abstraction layers. Each layer implements a particular feature and consists of program parts native to some programming language (e.g. classes, methods, functions, templates, mixins, etc.). A layer is a refinement of some other basic application layer. Stacking layers onto each other yields a complete application that contains the features implemented by the respective layers. C++ templates [Cza98] and Java mixin-layers [SB98] are commonly used to implement GenVoca layers. In each layer, mixins and plain classes are the basic abstraction mechanisms.

Subject-oriented Programming (SOP) [HO93, OKK+96] is a technique where software is built through the composition of subjects. Each subject is a collection of program parts and the composition merges appropriate parts so as to build the resulting program. There are many composition rules and they are described separately from the subjects. This contrasts with the mixin-implementation technique of GenVoca, where only one kind of composition technique is used to compose program parts.

Integrative Composable Generators (ICG) [Bri05] produce software with a set of program generators. A program generator essentially consists of multiple generative programs that each produce a well-defined part of the generated program. Some of the program parts are exposed through an integrative composition interface and can be integrated in another generated program. Such an integration is specified using an integration specification. The program generators can automatically adapt their generated programs to achieve an integrative composition specified by an integration specification so as to avoid composition conflicts. These conflicts are automatically detected by the generative system and force the program generators to resolve them.

3.6.1 Data Structures

Structure and Organization A basic compositional generator offers a predefined set of operators to compose a target program with a series of smaller programs or program fragments. In essence, these operators also define the source language. In advanced compositional generators (such as [Bri05]) the source language is a configuration language for a set of target program fragments.

Construction Unlike transformational approaches, program fragments are constructed upfront by using the mechanisms and abstractions provided by the target language. Each program fragment is, as the name already suggests, a partial program specification. The only restriction is that each program fragment must be syntactically correct and semantically sound. More concretely this means that a program fragment cannot omit expressions that are essential or mandatory. For example, an invalid program fragment is a function call with fewer arguments than its formal parameters. Note that, this does not imply that program fragments have to be complete.

The degree of parameterization in these approaches range from zero, via fixed to unconstrained. The program fragments in subject oriented programming cannot be parameterized. The mixins of GenVoca only parameterize the static super class of a class, while C++ templates parameterize over any type that is used in the program fragments. Integrative composable generators can parameterize any part of a program fragment.

Modification Modification of program fragments is performed by composition rules. When chosen carefully, the internal consistency and correctness of the program fragments involved in a composition can be maintained. *Composition rules involve detailed knowledge about the inner workings and parts of a program fragment.* This hampers future evolutions and increases maintenance costs. It is particularly the case for techniques like SOP where a large number of rules are available which are separately described from the actual subjects. Even though this separation promotes reuse of rules and subjects, it endangers consistency and correctness as the subjects have no control over them.

Techniques, like GenVoca and ICG, have a very restricted set of composition rules which need to be tightly coupled with program fragments. For example, a GenVoca mixin class has a built-in composition technique, i.e. parameterizable superclasses. Since the composition technique is part of the definition of a mixin class, a higher level of consistency and correctness can be ensured. Note that additional rules are still required to avoid semantic conflicts. Controlled modification taken to the extreme is done in ICG. Modifications to generated programs or program fragments can only be accomplished by talking to the integrative composition interface of the generator. To obtain a high degree of reusability, *a*

well performed upfront design of the program domain of a generator is necessary so as to anticipate the necessary compositions and conflicts with other generators.

Deletion Deletion or cancellation of program fragments is not possible. When deletion of some part of a program fragment is necessary, it should have been separated from the beginning into an another program fragment.

3.6.2 Transformation Modules

A transformation module is a program fragment generator. Similar to templates, compositional generators organize a language implementation according to the structure of a target program.

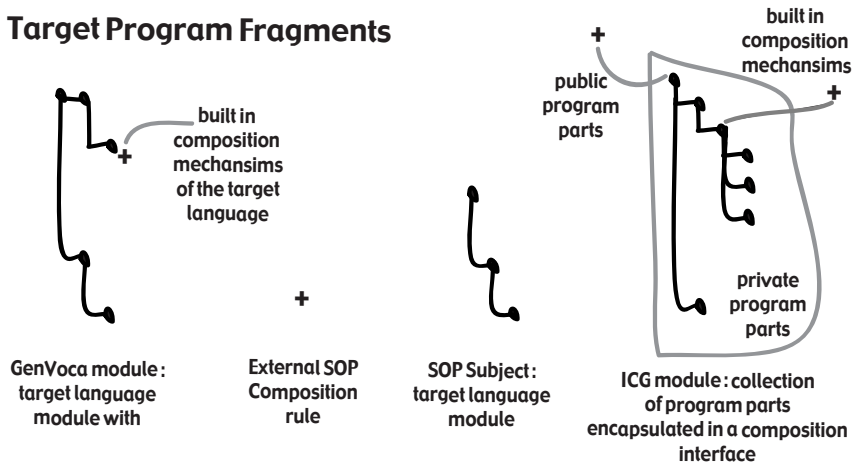

Figure 3.16: A schematic overview of a heterogeneous compositional generator

Identification The ability to uniquely identify the program fragments to compose is indispensable. It furthermore defines to a large extent the way these formalisms/systems operate: program fragments are statically identified and composed. As a result the granularity of the program fragments is usually restricted to named program entities like classes, or sometimes individual methods in class-based object-oriented languages.

Scope A source language can only be defined in the advanced compositional techniques. As the modularization in compositional generators is focused on the target program, compositional generators can be classified as target-driven approaches. In these approaches, transformation modules have unlimited and

global access to the entire source program. ICG, offers a powerful query technique to consult and retrieve the necessary information.

In most compositional techniques the target scope is limited to a single program entity like a class. Recently however, in ICG the target scope is virtually unlimited and can include various program entities.

Scheduling Scheduling is determined by composition rules that combine various program fragments into a single working program. In most cases composition is scheduled through the order of the compositions that combine program fragments. However, in more advanced approaches such as ICG, scheduling is controlled by an underlying constraint network.

Global-to-Local Compositional techniques can be classified as a target-driven approach. For those in which a source language can be defined, we refer the reader to our discussion of template-based approaches in Section 3.4.2.

Local-to-Gobal The implementation of local-to-global transformations is explicitly supported in program compositional generators. There are two major challenges to implement local-to-global transformations: the distribution and the integration of nonlocal results. Compositional generators have a *symmetric* data model: all results are potentially nonlocal results, as every result might need to be combined with another result.

Despite the explicit support of local-to-global transformations, compositional approaches still share the problems of template-based approaches (see Section 3.4.2).

Although the composition is statically described - through identifying the fragments which need to be combined - there is still a need to distribute code fragments. Recall that compositional generators operate on coarse-grained program fragments consisting of several smaller program parts. The combination of these coarse-grained fragments usually requires a more fine-grained composition mechanism to handle all the smaller program parts. Hence, the distribution problem in a compositional generator is situated within composition.

Simple compositional generators avoid that problem by constraining or predefining the compositions that can be expressed. As such, compositions can be facilitated and complex distributions can be avoided. Examples of such an approach are C++ templates and GenVoca generators. The latter offers parameterizable superclasses as the only way to compose programs fragments. The semantics of a composition totally relies on the inheritance semantics of the target language. Other compositional generators face the distribution problem upon composition. One of those techniques is SOP. SOP reduces the complexity by offering a set of language dependent basic operators. These operators are intelligent and take into account the complexities upon combination. Yet another kind

of compositional generators (such as ICG) do not restrict the operators but force a design in which individual program parts are made identifiable. Concretely, the integrative composition interface of ICG exposes a selected set of program parts of a generator which may need to be identified by other generators.

The combination of results in composable generation is governed by composition rules, composition specifications or composition operators. Composition is actually a third-party contract. There are the two program fragments that need to be composed and there is an external actor dictating that composition. *None of the composition approaches assigns the proper set of responsibilities to each party.* There are basically two extremes. In SOP, composition is defined and executed by externally defined rules. The parties which are being composed are not involved in this process. In other words, the program fragments are treated as data and serve as input for the external rules. At the other extreme of the spectrum lies ICG. Program generators, which produce the program fragments, offer through an integrative composition interface not only the parts but also the rules (relationships) which can be established among these parts. The program generators must effect these relationships during the generation and combination process.

Strengths and Weaknesses

\+ Compositional generators advocate a symmetric model.

\+ Compositional generators explicitly support integration.

− Compositional generators do not balance the composition. They either treat the program fragments or the composition as data.

− Compositional generators do not define an actual source language

− Compositional generators statically define the composition

3.7 Ad-hoc Approaches

Ad-hoc language implementations consist of small tools or libraries which can be used to implement a single phase of a language implementation. In order to write a compiler for a language, these separated phases are combined by interfacing with the various tools or libraries. Interfacing is performed in a general purpose language and requires significant insight in the innerworkings of the tools and libraries. Ad-hoc approaches make use, are defined and operate with concepts of general purpose languages. The language implementations are thus also written in terms of general purpose languages and not in a dedicated system or formalism.

Ad-hoc implementations are a heterogeneous group of systems. We selected five systems: Delegating Compiler Objects, Intentional Programming, Jakarta

Tool Suite (JTS) and Functional Programming. Delegating compiler objects demonstrate the need for an alternative architecture for compilers and illustrate that this attempt in its preliminary stage raises more questions than answers. Intentional programming is a language-less system such as macros and demonstrates the use of a first class representation of transformation modules. JTS is a prime example of a compiler using object-orientation to construct extensible languages. Functional programming has a long history in compilation and interpretation of language, offering generic traversals and monads.

3.7.1 Delegating Compiler Objects, JAMOOS and TaLe

The main goal of delegating compiler objects (DCOs) [Bos97] (see Figure 3.17) is to achieve modular, extensible and maintainable implementations of compilers. The authors claim that

> ...existing approaches to compiler construction do not provide the features required for application domain languages and extensible language models. These problems are related to the complexity of compiler development, extensibility of compiler components and reusability of elements of an existing compiler [BD].

Delegating compiler objects provide an alternative for the conventional, monolithic approach of compiler implementations. Bosch [Bos97] suggests delegating compiler objects (DCOs) as a means for DSL implementation. The philosophy of DCOs is that next to the functional decomposition into a lexer/scanner, parser and code generator, another decomposition dimension is offered, i.e. structural decomposition.

Structure and Organization The source program and the generated code (target program) are stored in a graph. It contains the abstract syntax trees of both source and target programs. Additional edges establish other relations like `change` and `instantiation`, which turn the tree structures into a graph. Change relationships relate how a program fragment affects another program fragment. Instantiation relationships relate an program fragment to its DCO.

Transformation Modules A transformation module in DCO boils down to a small compiler for a portion of an actual language. It is unclear what constitutes a DCO, in other words what is a typical DCO and how we recognize it. A DCO defines the syntax and contains the translational semantics.

Syntax DCOs contain multiple lexers, multiple parsers. Portions of grammars are reused through reuse of parsers i.e. by delegating to parsers of other DCOs. The reusing parser can override/add productions of the reused parser. The reused

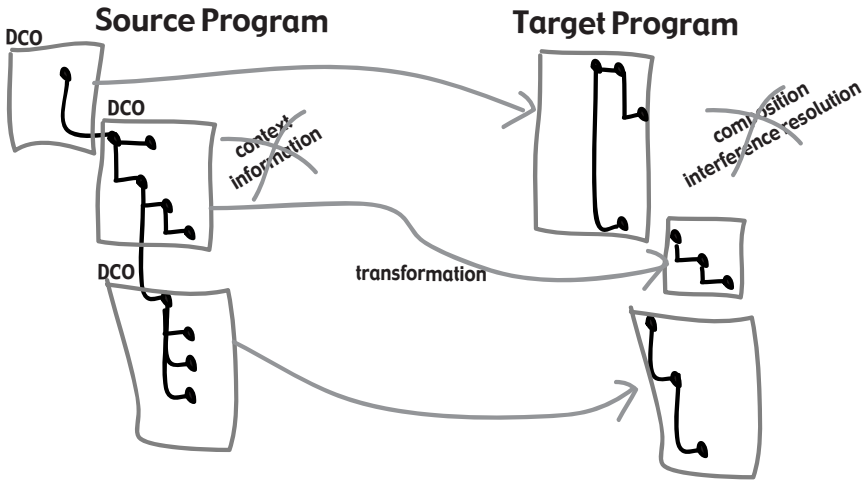

Figure 3.17: A schematic overview of a DCO

parser has access to the reusing parser to be able to give priority to the overridden and added rules. The nonterminal and terminal symbols are globally known to every DCO, and are used as a means to connect DCOs.

DCOs focus mainly, during the division of a monolithic compiler, on the parser and lexer phases. We use deliberately use the term 'division', because the individual DCOs are merely grammar and compiler parts which are put into separate text files. More precisely, DCOs explicitly reference each other by sharing a set of terminals and nonterminals.

With respect to the parser and lexer subdivision in DCOs, researchers working on JAMOOS made an interesting observation. JAMOOS [JY01] and TaLe [JKN95] dwell on the correspondence between OO types and BNF grammars. Their motivation and main objective was to carry out a unification of the concepts of language processing and OO programming to the fullest possible extent. They observed that: "It is not possible to restructure the abstract grammar. This limitation proved to be annoying in the actual language definition." [JY01]

Effect DCOs contain besides multiple lexers, multiple parsers also the ability to execute compilation. Traditionally code generation follows a centralized approach in which a function consumes the entire graph and produces a result. The code generation process of DCOs follows a distributed approach where each parse node is capable of compiling itself into a piece of the target (or output) graph.

Global-to-Local and Local-to-Global One of the major drawbacks, due to the distributed approach is the lack of context information. No solution is pro-

vided to compensate for this lack.

Another problem are interferences between changes and additions of multiple source graph nodes to a common output graph node. The interferences are categorized into 4 categories: shared unordered parent construct, shared ordered parent construct, shared unordered construct, shared ordered construct. How these conflicts are resolved, and the amount of effort required to do so, is not further detailed by Bosch et.al. [BD].

3.7.2 Intentional Programming

Intentional programming (IP) [WdMS+01, Sim95b, Sim96, ADK+98] is an early illustration of what is known as Fowlers Workbench. Recall that the emphasis in workbenches lies on the advanced integration of tool suites and domain-specific languages. The primary operating mode in those environments is a powerful graphical user interface which is highly coupled with the idea of an open-ended programming language. Programming in these environments involves the direct manipulation through a (graphical) user-interface of an abstract data structure which represents the program. The data structures are instantiations of program constructs which can be chosen and extended at will. Intentional programming is the embodiment of that idea.

Languages in IP contain an open-ended number of language constructs called intentions. Each *intension* defines a (forward) transformation that implements the semantics of the intention's language abstraction by generating program code for it, called a *reduction*. Intentions define much more than transformations. An interesting aspect of IP is that a source program is not represented as text but as active source, that is, as a data structure with behavior at programming time. This means that besides the definition of a transformation, each intention defines how it should be visualized in the program source (e.g. as a mathematical formula, a UI spec, etc.), how it should behave in the debugger, how it behaves in the version control system, etc. Each of these functionalities is defined by a separate method on the intention module, much like methods of classes in object-oriented programming.

There is not much technical information available about IP. As a result, a full and detailed analysis is not possible, and a figure similar to the other approaches we have described in this chapter could not be drawn.

Data Structures

Structure and Organization Source programs are captured in a graph. The program graph contains the abstract syntax tree of a source program and its corresponding target program, and various relations like dependencies and instantiations. One of the interesting things about the graph structure is the explicit link between the nodes that represent a concrete program and the intentions that

describe them. This link can be used in various ways. Yet there is little evidence that this first class representation of transformation modules is used to its full potential. What is known, is these edges are used by the graphical interface to depict the node graphically/textually on the screen.

The source graph is immutable. The target program is attached to the source graph. That part of the graph is mutable.

Scheduling A language in IP is defined as a set of intentions. Whenever new intentions are added, the schedule should be revised, requiring a detailed analysis of the influences between intentions. This obviously requires detailed knowledge of intentions. To overcome this, the reduction methods of all intentions in IP have to adhere to a few basic principles such that the transformation schedule can be determined by IP. The general premise is that the reduction method of each intention can assume that the entire source/target graph is already in its final state, except for the changes to be performed by the reduction method itself. This does not imply that the source/target graph may not change. If the information in a particular node is changed (e.g. by adding a new link to the node), all methods that were already invoked on that node are re-executed and the results are compared with the previous executions. When the results have changed, the system *rolls back* to a point in the reduction process where the methods were not yet invoked and tries invoking the methods in a different order. This is monitored and enforced by the IP system. Again more details are missing.

Global-to-Local and Local-to-Global One of the core technologies of intentional programming is attribute grammars with forwarding. For more details on how global-to-local and local-to-global transformations can be implemented we refer to a discussion in Section 3.5.2. In addition, the intentional programming system R.5, provides language features to deal with relative information more easily. When a child intention requests information from its parent intention, the parent can use the 'who is called' parameter to distinguish between several kinds of children or their properties.

3.7.3 Jakarta Tool Suite

Jakarta Tool Suite (JTS) [BLS98] (see Figure 3.18) or Jak is a DSL building environment that is based on GenVoca generators. Each DSL-extension to a language is a GenVoca component that can be composed with a base language. From the grammar, a layer (mixin) is generated with a class for each nonterminal (left-hand side of the productions) of the language. This extensibility is what makes JTS interesting.

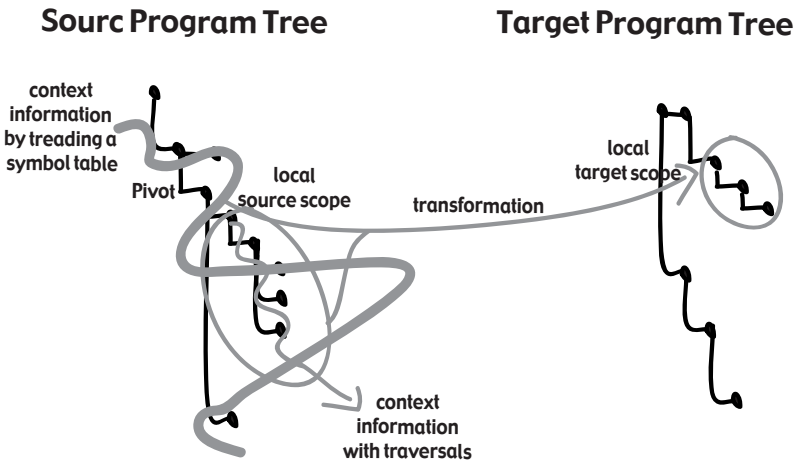

Figure 3.18: A schematic overview of JTS

Data Structures Source and target programs in JTS are represented by two separate abstract syntax trees. The target tree is a mutable structure, allowing changes to be made to it by several transformations.

Transformation Modules Transformations are written inside the classes which correspond to the nonterminals of a language. As such, linking the semantics of a language with its syntax. Transformation starts with the invocation of the `reduce2java(...)`-method of the class corresponding to the toplevel production. The parameter in this method may serve as a vehicle to pass around information from one location to another, providing of course that the information flow matches the recursive descent traversal of the source code by the generator.

Global-to-local JTS offers two mechanisms to implement global-to-local transformations. The first mechanism is based on parameter passing through the `reduce2java(...)` method. Parameter threading is one of the most rudimentary mechanism to pass around information. When the parameters are reduced to a single one, and the parameter value acts like a container to store and retrieve information, we end up with a *symbol table*. Symbol tables are one of the first strategies to transport data from one location to another (see Section 3.7.4). Note that, symbol tables may need to be constructed in a separate phase prior to transformation, in case information flows from children to parents.

The second mechanism offers some basic primitives for tree traversals. Code written using these traversals contains very detailed information about the structure of the traversed tree.

Local-to-global Global-to-local transformations are not supported in JTS. One must revert to handcrafted solutions, similar to those used in rewrite rules (see Section 3.1.2).

3.7.4 Functional Languages

Functional programming has a long history in compilation and interpretation of languages. Most of them are equipped with a strong typing system and enforce a strict separation between functional modules. For the purpose of this dissertation functional languages are important because of their strong typing, generic traversals and monads (depicted in Figure 3.19).

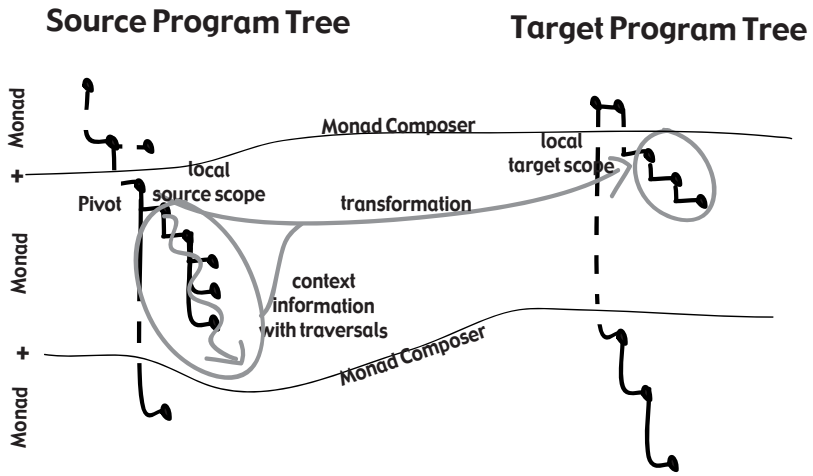

Figure 3.19: A schematic overview of a functional language implementation using generic traversals and monads.

Structure and Organization Both source and target program are represented with abstract data types. As all data structures in a functional language, the target program is immutable. Hence, upon the construction of a program fragment, all its parts must be available. Needless to say that language implementations in functional languages are very similar to attribute grammars in that respect. Similar to graph and tree-based rewrite rules, the abstract data types do not have a reference to their parent, and thus complicate information retrieval from arbitrary locations.

Transformation Modules A transformation of a program is implemented with a recursive function that computes a new target program fragment by com-

bining the results obtained from the application of that function to the parts of source program fragments. We refer to this function as the transformation function.

Global-to-Local and Local-to-Global Functional language implementations do not distinguish between global-to-local transformations or local-to-global transformations. Both are resolved by changing the transformation function such that information flows between its successive applications.

In order to access information which is external to a program fragment, the signature of the transformation function needs to be changed. This way, information can be passed from one application to the next. Changing the signature is an invasive operation and thus expensive operation. In an attempt to limit the impact of this operation, one usually uses a symbol table instead of opting for a parameter per external information request.

Whenever information needs to flow against the recursive descent of the transformation function, that information needs to be computed via an additional traversal initiated at a common ancestor. This not only requires to change the transformation function in various places, but also requires some generic traversal over a typed tree. For this, implementations require a metastructure (e.g. container - element structure) to facilitate these traversals and queries. Some require developers to directly use these metastructures. The metastructures are tree presentations containing generic nodes and leaves. With the use of metastructures the grammar encoded in the types is lost. Due to this loss, syntactically incorrect programs which can be constructed are not checked by the compiler. Other approaches such as the strafunski [LV03] library for Haskell generate the metastructures and as such provide typesafe generic traversals. Others [GdM03] use a dual representation and use reflection to traverse and use base types to implement the traversal actions.

Although the symbol table works it is only a part of the solution. Suppose there are several information flows, then we need several symbol tables or a symbol table of symbol tables. Anyhow, this change will need to be reflected in the semantics of every language construct. Another reason is that every transformation still needs to be aware of the need for a symbol table. The solution for this problem is a mechanism that would allow you to control the combination of functions a.k.a. monads.

Monads [Wad92] allow the programmer to build computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required [New]. As such, the recursive application of the transformation function can be controlled by a monad. One of the prime examples of monadic programming is the state monad. The state monad propagates state information from

one function application to the next one. This way, the translational semantics of language constructs can be made oblivious to state. The challenge with monads lies in their combination. A monad transformer is needed, in order to compose monads. Unfortunately and as expected, the construction of monad transformers for complex monads is not trivial.

Monads are a vehicle, they do not provide a specific suite of operations and an abstraction to construct language. The generic nature of monads is both its strength and its weakness.

3.8 Discussion

In the discussion of the various kinds of LDTs the focus lies on the question to what extent a modularized language implementation can be constructed. For each LDT, we have detailed the data structures, investigated how transformation modules identify what to transform and their scope, how transformations are scheduled, how information is acquired to perform a transformation, and how translational semantics is expressed for the various classes of transformations. These are our findings.

Architecture As we said earlier, not all LDTs strictly adhere to the typical compiler architecture. To a certain degree LDTs attack the monolithic nature of the typical compiler architecture. In increasing order of defiance: Rewrite rule systems are a prototypical implementation of the typical compiler architecture. All the phases merely process data, changing it, adapting it, to yield the target program. Compositional generators have a fixed vocabulary, fixing the parser and lexer. Attribute grammars give each production rule of each nonterminal the necessary attribute definitions to participate in global analysis and to produce the target program. Delegating compiler objects divide the monolithic architecture into a large a set of smaller compilers. Macros and intentional programming designs languages using an open-ended set of macros or intentions.

The remainder of this discussion is structured similarly to the analysis of the individual LDTs. We start by sketching the various data structure tradeoffs and their operations, and we end with an overview of techniques and tradeoffs to implement the various classes of transformations.

Datastructure Implications Programs are typically stored in abstract syntax trees. However, more expressive representations such as graphs open opportunities to corepresent other structures and information. One example of an alternative structure is presented in Intentional Programming (Section 3.7.2), where intentions are explicitly linked to the nodes representing source and target program.

Tree-based rewrite rule systems use a minimal tree representation only storing the recursive composition of terms. All other systems link terms/nodes via a bidirectional navigable link. Although performance is lost, the ability to access the parent facilitates the retrieval of information considerably.

The representations for source and target programs range from inert terms to flexible record-like structures. At the one extreme, terms cannot be changed once they are produced. In this setting, a local-to-global transformation must be implemented as a global-to-local transformation where upon construction, all the parts are be combined. Because the protocol to combine those parts is highly dependent on the semantics of their producers, the semantics of a local-to-global transformation gets distributed along several modules. As it is in general impossible to predict which transformation will be triggered and thus create a particular target program fragment, it is difficult to assign the creational responsibility for that construct to single transformation module. Implicit creation in graph rewrite rules (see Section 3.5.1), or combination rules in compositional generators are a step forward towards a more fitting implementation strategy for local-to-global transformations. A similar (but limited) effect can be achieved by rewriting. Terms produced by one rule, can be successively change by another rule.

At the other end, there are flexible record structures that can be partially constructed and modified. In that setting, local-to-global transformations change the partial data structures produced by other modules. However, careful consideration must still be taken in order to avoid data corruption and inconsistencies with the rest of the code. Those considerations are not guaranteed as data containers are passive and not encapsulated. Representations are thus primarily chosen to structure data. An interesting approach is taken by ICG, where program parts participate in a constraint network to enforce consistency.

LDTs with the most flexible data structures work with one data structure initially representing the source program and gradually changing it to the target program. LDTs with a more rigid data structure, store the source program and the target program in different data structures. While systems with flexible data structures have a uniform generation and optimization cycle, the systems with rigid data structures reduce scheduling problems.

Source program and target program are most of the time stored in separate data structures. Keeping them separate and stable ensures that a computation remains valid throughout the compilation process. Attribute grammars strictly enforce that policy by keeping data structures read-only.

Transformation modules In all LDTs transformation modules are fine-grained. The source and target scopes are restricted to minimize their dependencies. That tendency is even present in compositional generators and template languages. Compositional generators such as ICG offer small program fragments that can be subjected to change. Template languages equip their transforma-

tion modules with parameters in order to limit their source scope. As as result, the primary modularization focus on the target program shifts a bit towards a source focused modularization. Due to the restrictions on the source and target scope, additional facilities are necessary in order to implement local-to-global and global-to-local transformations. Such facilities cross the scope of the individual transformations.

Global-to-local transformations. Nearly every LDT offers explicit support for the implementation of global-to-local transformations. The support aims to facilitate the retrieval of information outside of the source scope of the transformation modules. Some systems increase their source scope. Success stories of such extensions have been made in attribute grammars extended with explicit copy rules and multiple inheritance, and in graph rewrite rules with variability in the subgraph structure. Besides explicit declarations, also implicit retrieval information schemes exists such as forwarding (Section 3.5.2).

Explicit retrieval of information is in most cases embedded in the transformation module itself, rendering them dependent on the existence of other transformations. Exceptions to this are ad-hoc approaches and attribute grammars. In those approaches, information can be provide without having to change the definition of transformation modules. There are also LDTs which allow modules to compute information for other modules. The most prominent examples are tree and graph rewrite rules. The drawback is that this approach generates more modules which enable the operation of other modules. This rapidly results in a complicated web of dependent transformation modules.

With graph rewrite rules, one can compute information and attach it to a node, which in turn can be used by another rewrite rule. Because of partial reasoning, computing additional information does not interfere with consumers (cfr. Section 3.2.1). The same can be achieved by tree rewrite rules, but there are more complications (cfr. Section 3.2.2).

Local-to-global transformations are rarely supported. Most LDTs regard them as the inverse problem of global-to-local transformations. Even in rewrite rules equipped with dynamic rewrite rules, the main idea is to capture information in one site and use it in another site.

Moreover an explicit distinction is made between local and nonlocal results. A nonlocal result requires that their is a local result present in which to add the nonlocal. Graph rewrite rules equipped with implicit node creation offer a symmetric model as one cannot predict which rewrite rule introduces the local result and which one will produce the nonlocal results.

The unification and combination rules are fixed in implicit node creation. Composition generators may offer some interesting perspectives in that regard. The composition is governed by specific rules and constraints, so in other words,

arbitrary changes to produced program fragments are not allowed. However, as composition rules are separated from the program parts, consistency and correctness are still not entirely guaranteed. The other extreme, where program parts are made up front responsible for the composition, clutters the translational semantics. However, these approaches are static and therefore have trouble taking into account complicated and context dependent control logic (Sections 3.1.4 and 3.6.2).

3.9 Conclusion

All LDTs divide a language implementation into a set of fine-grained transformation modules. The first observation is that it is not always clear how to relate these modules to language constructs. The second is that the modularity of modules is almost non-existing modules *implicitly cooperate* with one another.

This implicit cooperation is due to the fact that modules operate on a shared representation. This shared structure dominates the execution of transformations. Transformations are invoked to operate on a part of that shared structure and their results are subsequently processed by other transformations. As long as the transformations agree on the results produced by other invoked transformations this schema works fine. However, as soon as transformations are more complex i.e. they require more information than which is produced, or require different results from other transformations, or when transformations produce more results than others can use, then the modularity of language implementations decays. In these cases, one can either change the dominant structure by using intermediate representations or change the transformation modules. Changing the dominant structure increases the implicit cooperation, as new modules are added that enable the execution of other modules. Changing the transformation modules can also increase the implicit cooperation when transformations are polluted with new responsibilities in order to satisfy the needs of other transformations.

Clearly, there is need to change transformation modules so as to be able to execute in case a transformation does not agree with the results produced by other invoked transformations. However, only changes that do not add new responsibilities to transformation modules, but merely complement and adjust their semantics to their new environment, are allowed. Of the various LDTs we examined, only attribute grammars and ad-hoc approaches are capable of changing transformation modules while not increasing their implicit cooperation. Both kind of systems are implemented on top of general-purpose languages. We observe that they rely on the modularization and adaption features of these languages to define the transformation and to deploy the module in a concrete language implementation respectively. They also rely on flexible data structures to access information and to produce multiple results.

We encountered various mechanisms to reduce the amount of implicit coop-

eration. These successful mechanisms are scattered over the various LDTs.

- Tree-based rewrite rules offer traversals and dynamically scoped rewrite rules with rewrite strategies.
- Graph rewrite rules offers flexible data structures, implicit node creation and matching by morphisms
- Template-based approaches offer structure-shy queries
- Attribute Grammars offer implicit copy rules, explicit copy rules, templates, inheritance and forwarding
- Compositional Generators offer explicit composition rules and enforce consistency
- Delegating Compiler Objects (DCO) demonstrate the need for a new compiler architecture
- Intentional programming (IP) demonstrates the use of first class transformation modules
- Jakarta Tool Suite (JTS) offers symbol tables and uses general purpose language features to increase extensibility.
- Functional languages offer generic traversals, symbol tables and monads.

Chapter 4 introduces requirements that avoid the problems of the discussed related work, and describes a formal model for the modularization of language constructs that adheres to these requirements. Chapter 5 and 6 discusses a new LDT which implements our model. A validation is presented in Chapter 7.

Chapter 4

Modularization of Language Constructs in Language Implementations

Languages are grown by adding language constructs. In the previous chapter, we presented the contemporary language development techniques (LDTs) and concluded that they decompose language implementations into a set of implicitly cooperating modules. Furthermore, it is not always clear how individual modules relate to language constructs.

In this chapter, we define a model to modularize language constructs by applying the principle of separation of concerns. In order to modularize language constructs, we impose a number of requirements on the syntax and the translational semantics of the language. From this model we deduce a new language implementation design in which languages consist of three kinds of concerns: *basic language concerns* defining language constructs, *language specifications* defining interactions between basic concerns, and *special-purpose concerns* which define the mechanisms to implement interactions among basic concerns.

The first part of this chapter discusses the rationale behind our modularization model (Section 4.1), the impact on the implementation of languages (Section 4.2) and the requirements and challenges we impose on the design of a new LDT (Section 4.3). We conclude this part in Section 4.4 with an analysis of the separation of concerns deduced from our model. In the second part, we conduct in Section 4.5.1 an analysis regarding the inability to implement a language using modularized language constructs with the contemporary LDTs. In Section 4.5 we analyze the reasons for this shortcoming and distill remedies. We conclude this chapter in Section 4.6.

4.1 Modularization Model

The challenge in this dissertation lies in the modularization of complex translational semantics. At this point in the dissertation we define a complex transla-

tional semantics as a non-homomorphic translation or a global-to-global transformation.

We pursue modularization by applying the principle of separation of concerns (SOC). Separation of concerns is a powerful design principle for tackling the complexity of a system. At its heart lies the divide and conquer strategy, which has proven to be a very effective way to solve problems. The technique further extends the divide and conquer strategy with the goal to decompose a system into smaller modules each containing a single aspect of a subject.

Dijkstra [Dij76] primarily regarded separation of concerns as a technique to order his thoughts and efforts, without a concrete design technique in mind. As a result, techniques to separate concerns need to be conceived before the principle can be used to structure implementations.

Separation of concerns allows developers to solve a given problem by focusing on individual concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as adaptability, maintainability, and reusability.

We decouple and parameterize the implementation of language constructs, in order to separate and modularize the language along the dimension of its constructs. Whenever the language changes, the language constructs involved can be modified and later on recomposed into a new version of the language. The resulting incremental language design approach has been described by Gholoum [Ghu06] as a natural way to engage in its development. It allows us to easily restructure a language and its compiler, a need which has been observed in other related compiler decompositions like JAMOOS [JY01] and TaLe [JKN95] (see Section A.1).

4.1.1 Setting the Stage

In order to present the formalization of the modularization of a language implementation according to language constructs, we must briefly introduce the definition of programming languages and its semantics defined in terms of another language.

Programming Languages

Noam Chomsky defined a formal language L as a set of finite-length sequences (sentences) of elements (vocabulary) drawn from a specified finite set of symbols. We chose to specify a formal language using the type-2 grammars of the Chomsky hierarchy, as it is the most widely used type of grammar.

4.1. DEFINITION. *A context-free grammar [Cho56] is a tuple* $G = \langle T, N, S, P \rangle$ *where*

T *is a set of terminal symbols,*

N *is a set of nonterminal symbols (syntactic categories),*

S *is an element of N called start (or initial) symbol,*

P *is a set of productions, i.e. rules of the form* $A ::= \alpha$*, where A is an element of N and* α *is a string of terminal and nonterminal symbols, i.e. an element of* $(N \cup T)^*$.

The representation of the sentences of a language is an important consideration for the formalization of the semantics of a language. There exists a fairly standard compiler decomposition into computations over trees providing an abstract representation of a program [KW94]. In this dissertation we adopt trees as a means to represent programs, whose structure is formally defined by a grammar.

4.2. DEFINITION. *A language L is a set of phrases defined by an* ***algebra*** $\langle \mathbb{E}, \mathbb{F} \rangle$ *[Jan96], given a grammar* $G = \langle T, N, S, P \rangle$*. It consists of a set* $\mathbb{E}$ *containing the elements of the language, and a set of functions* $\mathbb{F}$ *defined on the set* $\mathbb{E}$ *and yielding values for* $\mathbb{E}$*.*

The set of phrases is a subset of the freely generated terms comprising a function symbol $F \in \mathbb{F}$ *followed by a number of arguments* $\alpha_i \in \mathbb{E}$ *(i=1...n), n being the arity of F. The terms represent the constructs in the language.*

For all $p \in P$*, of the form* $nt ::= \alpha_{p1}, ..., \alpha_{pn}$*, with* $nt \in N$*, a corresponding function* $F_i(\alpha_{p1}, ..., \alpha_{pn})$ *is defined.*

The set of primitive values $L_D \subseteq \mathbb{E}$ *is defined as the union of the domains of the terminals* $T : L_D = D_{t_1} \cup ... \cup D_{t_n}$*, where* $t_i \in T, 1 \leq i \leq size(T)$*.*

The sentences (programs) of the language L are a recursive combination of its phrases bound by the rules stipulated in the grammar of the language L and by the set of primitive phrases of L_D.

Translational Semantics

Let L_1 be the source language and L_2 the target language. We define the translational semantics of the source language L_1 in terms of the target language L_2 using denotational semantics. The denotation maps syntactic objects of language L_1 into objects of language L_2. We use denotational semantics because of its familiar and powerful mathematical objects, i.e. functions.

The domains of valuation functions may be constructed from primitive domains using products, functions, and lists. Primitive domains are

formed by adjoining two finite or denumerable sets such as $\{true, false\}$ or $\{..., -2, -1, 0, 1, 2, ...\}$, with two special objects $\top, \bot$ called *top* and *bottom* respectively. $\bot$ represents completely undetermined information[1]. We use it to denote syntactically and/or semantically invalid expressions. $\top$ represents overdetermined elements. The top element is not used in our formalization.

4.3. DEFINITION. *The denotational definition of the translational semantics of a language L_1 into L_2 is defined as follows:*

- *The syntax of the languages L_1 and L_2 is defined through their corresponding context-free grammar definitions G_1 and G_2.*
- *The semantic algebra is the target language L_2.*
- *The valuation function C defines the translational semantics, which we refer to as the compilation function. $C : L_1 \rightarrow L_2$*

The goal of our model is the modularization of a language implementation into its language constructs. Language constructs are defined as the syntactical constructs of a language having a distinguishable semantics with respect to the other language constructs. In our formalization we do not formally define a separate notion for a language construct. A language construct can be any valid phrase of a language ranging from a terminal $t \in L_D$, via a single function $F \in \mathbb{F}$ to an arbitrary nested phrase like $F(I(i_1, ..., i_n), ..., J(j_1, ..., j_p))$.

Language constructs often strictly correspond to the non-terminals of a grammar of a language. More precisely, each non-terminal and its set of productions defining it represent a language construct. However, such a definition merely shifts the problem to the question what exactly constitutes a non-terminal and the complexity of the production rules defining it. So, in our opinion, what constitutes a language construct should be left to the language designer.

4.1.2 Phenomena Described by the Model

In order to modularize a language implementation according to its language constructs we must impose requirements on the valuation function C to ensure that the definition of the syntax and translational semantics of each language construct is solely concerned with that construct. We prohibit the presence of any kind of dependency among the language constructs ranging from a direct reference to other constructs, to any implementation decision that is imposed by another construct.

[1]The $\top, \bot$ and bottom elements are also necessary to express the denotation of recursive expressions. The denotation of recursive expressions is formulated as a the fixpoint in a series of approximations of its semantics. For this, the domain must have a partial order, where $\bot$ is the smallest element and $\top$ the largest element.

The five requirements address the following phenomena. The list of phenomena is the result of an analysis of syntax and translational semantics descriptions found in contemporary LDTs (see Chapter 3).

P0 The translational semantics of a language construct does not yield a correct phrase in the target language.

P1 The translational semantics of language constructs does not compose.

P2 The translational semantics of a language construct requires information external to the language construct.

P3 The translational semantics of a language construct exercises an effect on the translational semantics of other language constructs.

P4 The syntax of a language construct refers to the syntax of other language constructs.

The phenomena and their corresponding requirements are organized into three logical groups: program representation (P0), translational semantics (P1-P3) and syntax (P4). We discuss them in a different order allowing us to better structure our explanations.

- Phenomenon P1 is based on the principle of compositionality [Mon74, Jan96], stating that the translational semantics of a phrase is formulated in terms of the translational semantics of its parts. Requirement R1 enforces that the modularization is maintained even in cases when modularized translational semantics of language constructs does not compose (see Section 2.1.3).

- Phenomenon P0, P2 and P3 are based on the fact that the translational semantics of (expressive) language constructs cannot be expressed by a mere one-to-one mapping [Fel91]. Hence, in general, we cannot expect that the translational semantics yields a correct phrase in the target language. This phenomenon P0 is addressed by requirement R0. More complex mappings in translational semantics have been classified in [vWV03]. In essence, one distinguishes mappings which lack information to produce their translational semantics (P2) and lack the ability to affect the translational semantics of other language constructs (P3). The modularization of the former is addressed by requirement R2, and the modularization of the latter by requirement R3.

- Phenomenon P4 is based on the observation that contemporary context-free grammar description formalisms such as BNF [BBG$^+$60], JAMOOS [JY01], DCO [Bos97], SDF [HHKR89] formulate the syntax of a language construct by directly referencing the syntax of other language constructs.

With these phenomena we can define the term complex translational semantics more precisely.

4.4. DEFINITION. *Translational semantics is complex when it exhibits either of the four phenomena P0 to P3.*

Note that phenomenon P4 is not part of the above definition as P4 is about the syntax of language constructs and not about their semantics.

The remainder of this section is divided into two parts. In the first part, we start by motivating and informally describing the requirements R0 to R3 that are imposed on the valuation function of a language. Afterwards, we formally introduce modularized valuation functions. In the second part, we detail requirement R4 to modularize the grammar specification. We conclude this section with a summary of the five requirements.

4.1.3 Compositionality Requirement (R1)

As the goal of our model is the modularization of a language implementation into its language constructs, valuation functions need to exploit the structure of the phrases of a language. By exploiting the structure of the phrases, the semantic definitions can be structured according to the syntax of the language, such that individual language constructs can be analyzed and evaluated in relative isolation from other constructs. Formally, this is achieved by imposing R1 which requires that the valuation function is compositional (Tennent [Ten91]).

Compositionality implies that the valuation of a compound expression combines the valuation of its parts according to a production (syntactic rule) e.g. `if` statements consisting of a condition, a true and else expression are evaluated using the semantics of the condition, the true and else expression. Suppose that a phrase E comprises the parts E1 and E2. Then compositionality states that the semantics C(E) of E can be found by finding the semantics C(E1) and C(E2) of its subparts, and combining them (according to some semantic rule). The property is recursively defined for all phrases E of the language.

The translation of an `if` expression written in L_1 to a series of instructions of a byte code language L_2 can be defined by a compositional valuation function C as follows:

$$C(\texttt{N}) = \texttt{CONST N}$$
$$C(\ \texttt{N1>N2}\ ) = \texttt{CJUMP(>,}C(\texttt{N1}), C(\texttt{N2}), \texttt{labeltrue}, \texttt{labelfalse)}$$

$$
\begin{aligned}
C(\texttt{if(T1, T2, T3) }) = {} & \mathrm{C}(\texttt{T1}) \\
& \texttt{labeltrue:} \\
& \mathrm{C}(\texttt{T2}) \\
& \texttt{JUMP(labelend)} \\
& \texttt{labelfalse:} \\
& \mathrm{C}(\texttt{T3}) \\
& \texttt{labelend:}
\end{aligned}
$$

where

$$N1, N2, T1, T2, T3 \in E_1$$

The above valuation function is defined for the basic phrases of the source language by using the translational semantics of its parts i.e. for each terminal and for each of the functions of the language algebra L_1. Firstly, a `number` is mapped to a `CONST` operation. Secondly, a `comparison` operation is mapped to a conditional jump (`CJUMP`) expression, which jumps to a label in case its condition evaluates to true and to another label in case its condition evaluates to false. Finally, an `if` is mapped to a series of intermediate instructions placing labels and jumps between its branches. Note that the labels to which the conditional jump jumps are hard-coded in the semantics of a comparator. As this is not desirable, nor usable in actual language implementation, these labels should in fact not be hardcoded. We opted not to do this yet, as this is another phenomenon in our model. In order to keep the discussion of the different phenomena clear, each phenomenon is discussed separately.

Compositionality and Homomorphisms

Compositionality in denotational semantics requires the presence of a homomorphism (Montague [Mon74]) between the algebra of syntactic representations and an algebra of semantic objects.

Intuitively speaking, a homomorphism from an algebra A to algebra B is a mapping which respects the structure of A. If in A an element is obtained by applying an operator f to $\bar{a}$[2], then the image of $f(\bar{a})$ under the homomorphism ϕ is obtained by applying an operator g (corresponding with f) to the images of a_i under ϕ.

$$\phi(f(a_1, ..., a_n)) = g(\phi(a_1), .., \phi(a_n))$$

4.5. DEFINITION. *A homomorphism between two languages L_1 and L_2 is applied as follows [Jan96]: Let A be the algebra $L_1 = \langle \mathbb{E}_1, \mathbb{F} \rangle$, let B be the algebra $L_2 =$*

[2] $\bar{a} = a_1, ..., a_n$

$\langle \mathbb{E}_2, \mathbb{H} \rangle$. *Let* $F \in \mathbb{F}$ *and* $H \in \mathbb{H}$*, then a valuation* C *is a homomorphism between* $\mathbb{E}_1$ *and* $\mathbb{E}_2$ *if:*

$$C(F(\mathbf{a_1}, ..., \mathbf{a_n})) = H(C(\mathbf{a_1}), ..., C(\mathbf{a_n}))$$

Compositionality and Granularity

Compositionality does not affect the granularity of a language construct, it merely allows us to split up the semantics of a phrase into a combination of the semantics of its parts. Suppose that for a given two phrases $F(I(i_1, ..., i_p))$ and $J(j_1, ..., j_p)$ their semantics are the same (equation 1). This assumption is reasonable as in many programming languages it is the case that different source phrases are in fact translated to the same target program. Due to the compositionality principle, the semantics of the phrase J is a new phrase H constructed using the semantics of its parts $j_1, ..., j_p$ (equation 2). Given equations 1 and 2, the semantics of the phrase F is thus a composition of the semantics of its nested parts $i_1, ..., i_p$. Clearly, compositionality also applies in the case where nested phrases are considered a single language construct.

$$\begin{aligned}
&(1) \quad C(F(I(i_1, ..., i_p))) &&= C(J(j_1, ..., j_p)) \\
&(2) \quad C(J(j_1, ..., j_p)) &&= H(C(j_1), ..., C(j_p)) \\
&(3) \quad C(F(I(i_1, ..., i_p))) &&= H(C(j_1), ..., C(j_p))
\end{aligned}$$

Compositionality and Composition

The meaning of a phrase (e.g. F and H in Definition 4.5 of the translation homomorphism) cannot solely be determined by its parts, as it depends on a rule that combines those parts into a phrase. Indeed, several phrases can be constructed from the same parts, yielding different meanings [PtMW90]. For instance, composing parts a and b in an assignment may lead to different meanings e.g. $a := b$ or $b := a$. Moreover, as compositionality does not affect granularity, the composition of parts can be more complex depending on the additional phrases that are being constructed e.g. composing the parts b, a, into the assignment with an increment yields $b := a + 1$. We encountered this in our example introduced in the beginning of this section. The $>$ construct is translated to a `CJUMP` containing two extra phrases `labeltrue` and `labelfalse`.

We explicitly expose the composition of parts into semantic definition functions $\mathcal{D}$. For each syntactical construct F of L_1 a semantic definition function $\mathcal{D}_F$ is a function of the meanings of the parts of F (and of the syntactical construct F).

Compositionality Conflicts

The definition function $\mathcal{D}_F$ of a syntactical construct F expects a number of specific target language constructs to be able to construct a syntactically and semantically valid equivalent semantic target language construct. Consider for example the following translational semantics of a `let`:

$$C(\texttt{let(T1, T2) }) = \texttt{MOVE } \text{C}(\texttt{T1}) \text{ C}(\texttt{T2})$$

where

$$T1, T2 \in E_1$$

A `let` is mapped to a `move` instruction and therefore expects that the translational semantics of both its parts yields two memory locations. However, the translational semantics of parts does not always suffice to construct its equivalent target language expression. We call such situations *compositionality conflicts*. In our example, a compositionality conflict arises when a `comparison` expression is used within a `let` between the translational semantics of a `let` and of a `comparator` expression because the translational semantics of a `comparator` expression yields a `CJUMP` expression instead of a memory location[3].

In order to resolve a compositionality conflict, the valuation function needs to anticipate the translational semantics of the subparts of a language construct and convert it into a suitable value. In our example, a `let` must anticipate a `CJUMP` and convert it into an expression that yields a memory location containing the boolean result (`-1` or `0`) of its condition.

$$C(\texttt{let(T1, T2) }) = \text{C}(\texttt{T2})$$
```
labeltrue:
STORE(R0 -1)
JUMP(labelend)
labelfalse:
STORE(R0 0)
labelend:
```
$$\texttt{MOVE } \text{C}(\texttt{T1}) \texttt{ R0}$$

where

$$T1, T2 \in E_1$$

Compositionality conflicts are not resolved by the definition functions $\mathcal{D}$, because that would require the explicit involvement of other language constructs and as such break their isolation.

[3]Changing the translational semantics of a `comparator` expression such that it yields a memory location is not a solution, as this translational semantics would then trigger a compositionality conflict with `if` expressions.

Compositionality and Strict Separation

Definition functions $\mathcal{D}_F$ consume a tuple $b_1, ..., b_n$ which are the translational semantics of the parts $a_1, ..., a_n$ of F. These are in turn tuple values representing program phrases of the language L_2. The $\mathcal{D}_F$ function consuming a tuple $b_1, ..., b_n$ has thus access to all the tuples nested within that tuple. Due to the compositionality principle, the nested tuples are the translational semantics of the language constructs $a_1, ..., a_n$. Consequently, there is a breach of isolation when $\mathcal{D}_F$ accesses the nested tuples of the parts $b_1, ..., b_n$ which have been produced by another language construct.

Requirement R1

4.1. REQUIREMENT. ***(R1)*** *To be able to define the translational semantics of a single language construct in isolation, a valuation function must be compositional using separate functions $\mathcal{D}$ denoting the definition of the translational semantics of a language construct such that the semantic definition of the translational semantics of a language construct:*

R1a *has no access to the source language construct, more precisely the semantic definition cannot access the parts of its language constructs but consumes a tuple of target language values obtained from recursively applying the valuation function to the parts.*

R1b *does not need to cope with compositionality conflicts*

R1c *should have limited access to the target language constructs it consumes, more precisely it should avoid directly accessing nested tuples.*

Compositionality and Expressive Language Constructs

The plain application of compositionality via a homomorphism does not suffice in the context of complex translational semantics for expressive language constructs. In the formalization by Felleisen [Fel91] (see Section 2.1.2) (expressive) language constructs could not be defined using homomorphism. For this reason, the translational semantics of languages requires a less restrictive application of a homomorphism. This is due to the fact that a homomorphism divides the semantics of a phrase in terms of its parts

- and therefore **reduces the amount of information or parts available to construct its semantics**. This case is handled by requirement R2 and requirement R0.

- by expecting a series of target language values and therefore **constrains the semantics of its parts on the semantics of the whole**. This case is handled by requirement R3.

4.1.4 Multiple Inputs Requirement (R2)

Translation by a homomorphism is very strict because a homomorphism requires that each language phrase F with a given arity n is mapped to a language phrase H with the same arity, consisting of the translated parts. However, the arity of phrase H may differ from the arity of phrase F e.g. in the translation of typed expressions sometimes their types are available but not used, or sometimes their types are not given and must be derived. In the construction of phrases with a different arity we distinguish between two cases: the arity of H is strictly greater or strictly smaller than the arity of F. The former is called an arity excess and the latter an arity dearth of H. In the case of arity dearth, the information in F needs to be selectively used or combined by $\mathcal{D}_F$ to yield a proper set of parts in order to construct H e.g. in the translation of variable declarations attributed with non-primitive or user-defined types the actual type does not influence the translational semantics. A less trivial situation arises when the arity of H exceeds the arity of F e.g. an operation does not contain the type of its operands which is required so that upon its translation the most appropriate implementation can be produced. In that case, extra information, external to F, needs to be obtained e.g. upon the translation of an operation, the types of its operands need to be computed. Consider for example the following snapshot of a compositional valuation of an functional language which supports the direct accessing and storing of information in computer memory:

$$
\begin{aligned}
& C(\texttt{let(}T1\texttt{,}T2\texttt{,}T3\texttt{)}) = \texttt{STORE}\ \mathrm{C}(\texttt{T1})\ \mathrm{C}(\texttt{T2})\ \mathrm{C}(\texttt{T3}) \\
& C(T1\ \texttt{+}\ T2) = \texttt{ADD}\ C(\texttt{T1})\ C(\texttt{T2}) \\
& C(\mathrm{ID}) = S(\mathrm{ID})
\end{aligned}
$$

where

$$
\begin{aligned}
& V \subseteq E_1 \text{ (storable values)} \\
& L \text{ (locations)} \\
& S = V \rightarrow L \\
& T1, T2, T3 \in E_1 \\
& \mathrm{ID} \in V
\end{aligned}
$$

The above equations define the translational semantics of a `let` expression to define variables as a `STORE` instruction, of an addition that uses variables as an `ADD` instruction, and of identifiers (`ID`) as a memory location. In theory,

one could define a function `S` which maps each identifier to a unique memory location, such that expressions which define and use the same variable use the same memory location. From a practical point of view, if at all realistic, this translational semantics performs poorly in terms of memory consumption. A more realistic translational semantics exploits the lexical scope of variables in order to reduce the memory footprint. For this, we need to keep track of the scope of variables in the `let` expression and reuse their memory locations in the translational semantics of variables as soon as their scopes are terminated. In other words, the translational semantics of variables require a symbol table containing a mapping between variables and locations. Clearly this table cannot be constructed solely using the `ID` construct and is thus external information required for the valuation of `ID`s.

As the meaning of a phrase may be codetermined by external factors, the domain and co-domains of the most general semantic functions consisting of the source language and of the target language respectively, in general do not suffice. Therefore, the conception of translational semantics has to be enriched. The domain and co-domains of valuations must be extended by composing the syntactical domains and the semantical domains with other domains. As such, information available in one construct can be passed to other constructs. The following code fragment illustrates the use of auxiliary domains in our example. A denotational function defining the meaning M of a sentence would look like: (more details can be found in [Ten76].)

Example $M : Exp \to S \to (E \times S)$
where

D	(denotable values)
V	(storable values)
L	(locations)
$S = L \to V$	(stores)
$E = D + V$	(the generic domain of all expressible values)

The symbol table S is passed from valuation to the next. As such, the `let` expression can allocate and deallocate memory locations in the symbol table which can then be used in the valuation of `ID`s.

Constraining Information

Requiring external information implies the involvement of other language constructs so as to obtain that information. With the prospect of separation of constructs, that solution should be avoided as much as possible. In order to maximize separation of the constructs, only essential information can be requested.

4.6. Definition. *In a mapping between a source language phrase say F to H, parts required to construct H are considered essential if they are necessary to reflect the semantics of the language construct F in a construct H.*

Preserving the Isolation of Semantic Definitions

In order to preserve the isolation of the semantic definition functions $\mathcal{D}$, a function $\mathcal{D}_F$ for a given construct F must not involve other constructs for the:

Identification of External Information The valuation function consumes a tuple which may consist of multiple language constructs. This is a direct violation of the separation of translational semantics of each language construct enforced by requirement R1. The successive applications of valuation functions are therefore prohibited from exchanging source language phrases among each other. Instead, the valuation functions must communicate derived values among their successive applications. We hereby strictly exclude the explicit involvement of other source language phrases in the semantic definition functions $\mathcal{D}$, in other words these functions can only consider one program phrase at a time.

Obtention of External Information Language constructs requiring multiple inputs can only be *defined*, the actual retrieval of the external information by depending or altering other language constructs cannot be executed by the semantic definition functions $\mathcal{D}$, as the execution would by definition involve other constructs.

Provision of External Information The valuation function must pass along external information to the computation of the semantic definition function $\mathcal{D}$. So in other words, if the valuation of a construct requires external information, the valuation of another construct has to provide it. We cannot allow that the valuation of other constructs is cluttered with computations for information which is only required by other constructs, as that would break the separation of the constructs. We therefore impose that semantic definition functions may only produce and provide additional information which is an essential part of the translational semantics of the source language constructs.

Requirement R2

4.2. REQUIREMENT. ***(R2)*** *To be able to depend on essential external information that could contribute to meaning while preserving the isolation, the semantic definition functions $\mathcal{D}$ of the language constructs:*

R2a *must be defined in terms of an arbitrary number of auxiliary domains. The extension of the co-domain is equal to the extension of the domain, as the auxiliary domains also serve as a carrier to pass information from one application to the next as the function computes the meaning of a phrase.*

R2b *only consider one language construct at a time.*

R2c *may only produce additional information which is an essential part of the translational semantics of the source language constructs.*

R2d *do not retrieve the external information*

The case where information is not essential, is handled by requirement R0 in Section 4.1.6.

4.1.5 Multiple Results Requirement (R3)

Semantics expressed using homomorphisms requires that the target language must have the same structure as the source language, so that a construct in one language is expressible via a local equivalent construct in the other language. These local constructs are then subsequently combined to form other constructs. The translational semantics of language construct is thus constrained to play a local role in the semantics of the overall program. So language constructs which affect the whole rather than just a local part can no longer be expressed. An example of such a construct is `callcc` [CFW85]. `Callcc` exposes the current continuation such that it can be explicitly used by the developer. However, in order to do that, a CSP transformation [Rey93] needs to be applied which potentially affects the whole program. By including the CSP as a part of the translational semantics of `callcc`, we can apply the CSP selectively [Nie01].

Translational semantics exercise effects on multiple phrases of the target program by injecting a new program fragment into a program fragment produced by another construct. We call these injected fragments *nonlocal results* or plainly *nonlocals*. In order to enable the production of those effects, we enrich the conception of the valuation function with the product of the target language algebra. As such, the translational semantics can produce multiple results.

Preserving the Isolation of Semantic Definitions

In order to preserve the isolation of the semantic definition functions $\mathcal{D}$, a function $\mathcal{D}_F$ for a given construct F must not involve other constructs for the:

Identification The nonlocal results can only be *defined*, the identification of the location where the nonlocal into the proper program phrase cannot be executed by the semantic definition functions $\mathcal{D}$. As the execution would by definition involve other constructs, this is prohibited.

Integration In analogy to the context information, if a construct produces nonlocal results another construct must consume them. However, we cannot allow that another construct be altered to consume those results and integrate them into their translational semantics. Only in case where these nonlocals are essential, exceptions to this rule are allowed. This ensures that the translational semantics of a language phrase is not cluttered with the additional task of integrating program fragments produced by the translation of another language phrase.

Requirement R3

4.3. REQUIREMENT. ***(R3)*** *A valuation function is able to exercise effects on the translational semantics of other constructs if:*

R3a *the domain and co-domains of the semantic definition functions are defined containing several times the target language domain.*

R3b *it only consumes external information or results which are essential for defining the translational semantics of a source language fragment.*

From requirement R3a and R3b we can deduce that the effect of the multiple results is either defined externally to the semantic definition functions, or the effect is provided by the semantic function of the producing language phrase. The effects can be passed on as external information by using a complex domain of so called integration functions Ξ. Ξ takes two target language phrases and produces a new phrase containing the integration of both input phrases.

$$\Xi : L_2 \times L_2 \to L_2$$

4.1.6 Representation Requirement (R0)

Recall from requirement R1 and R2 that for each syntactical construct its semantic definition function $\mathcal{D}_F$ is defined as a function of the meanings of its parts and

of essential external context information. In this section, we consider the case of non-essential information.

As the source L_1 and target language L_2 might be very different, we cannot assume that the translational semantics of a phrase of the source language is a complete language phrase in the target language. Recall the example given in Section 4.1.3. The translational semantics of the > construct is a `CJUMP` instruction. The `CJUMP` is a 5-tuple consisting of a conditional operator, its two operands, and a true and false label. If the condition evaluates to true, the instruction jumps to the true label, otherwise it jumps to the false label. We oversimplified matters a bit, as labels must refer to actual instruction indexes of the produced target program. To this end, an evaluation function, say M, must consume an additional value next to the language phrase denoting the index of the next instruction.

M(`CONST N`, index) = (`CONST N`, index+1)

M(`N1>N2`, index) = (`CJUMP(>,`M(`N1`,index),M(`N2`,index),⊥,⊥ `)`, index+1)

M(`if( T1, T2, T3)`, index) =
(ct1
ct2
`JUMP(`index3`)`
ct3,
index3)

where:
(index1, ct1) = M(`T1`,index)
(index2, ct2) = M(`T2`,index1)
(index3, ct3) = M(`T3`,index2+1)

where

$N1, N2, T1, T2, T3 \in E_1$

Using two parts of the > construct, only the first three parts of the `CJUMP` instruction can be computed as the > construct lacks information to provide the `CJUMP` instruction with a true label and a false label. The two labels are actually non-essential as the location where to jump is not necessary to reflect the translational semantics of the > construct. More so, the true and false labels are only known much later in the valuation process as the labels are part of the translational semantics of other expressions like `if`. It is only *after* the translation of its condition `T1`, its if-branch `T2` and its else-branch `T3` that the actual position of the true and false labels is known. Hence, these labels cannot be computed prior to the translation of the >. Therefore its translational semantics must yield an incomplete `CJUMP` instruction. For this, we impose that the representation of a program, given a definition of language L, must allow incomplete programs to be used as an intermediate state during the translation. Using the ⊥ element

of semantic domains, the translational semantics of a comparator expression can than simply ignore the labels.

As it is possible to produce incomplete target language phrases, the translational semantics of a construct can be defined without violating of their isolation and thus their modularization like:

- logic to handle every possible part that potentially may become a part of its equivalent target language construct,
- logic to provide every possible value which may be requested as a result of the above,
- dummy and non-essential semantic values that potentially may be required by the semantics of other constructs

Consistency

The co-domain of valuation functions is a complex structure representing programs in the target language. Care must be taken upon changing the translational semantics of constructs to fill in the missing parts, in order to ensure that the produced constructs remain syntactical and semantical valid. An example illustrating this is given below. For reasons of simplicitly we use label names instead of instruction indexes.

$$C\Big(\begin{array}{l}\texttt{if((a>b | !(b<c))}\\ \texttt{\quad\quad\& d > e), T2, T3)}\end{array}\Big) = \begin{array}{l}\texttt{CJUMP(>, }\mathrm{C}(\texttt{a})\texttt{,}\mathrm{C}(\texttt{b})\texttt{,w,z)}\\ \texttt{z:}\\ \texttt{CJUMP(<,}\mathrm{C}(\texttt{b})\texttt{,}\mathrm{C}(\texttt{c})\texttt{,labelfalse,w)}\\ \texttt{w:}\\ \texttt{CJUMP(>,}\mathrm{C}(\texttt{d})\texttt{,}\mathrm{C}(\texttt{e})\texttt{,labeltrue,labelfalse)}\\ \texttt{labeltrue:}\\ \mathrm{C}(\texttt{T2})\\ \texttt{JUMP(labelend)}\\ \texttt{labelfalse:}\\ \mathrm{C}(\texttt{T3})\\ \texttt{labelend:}\end{array}$$

The above equation defines the translational semantics of a boolean expression consisting of `and`, `or`, `not` and comparator operators as a sequence of conditional jumps and labels. The produced target language expression jumps to the `labeltrue` and `labelfalse` label if the whole boolean expression evaluates to true or to false respectively. Each conditional jump refers to a true and a false label in case its condition encoded in the jump evaluates to true or to false respectively. Notice that some conditional jumps jump, in case their condition evaluates to true, to the `labeltrue` label and some of them jump to the label `labelfalse`.

Hence, filling the true labels and the false labels is not a straightforward search and replace as the translational semantics of the boolean expression needs to be taken into account in order to deduce which positions of the conditional jumps need to be filled with either the `labelfalse` or `labeltrue` label.

Requirement R0

4.4. REQUIREMENT. ***(R0)*** *The translational semantics is defined in terms of the language algebra $L_2{}_\perp^\top$. Imposing this requirement has the following three consequences on program phrases:*

Partial (R0a): *The bottom value $\perp$ represents completely undetermined information. A phrase containing a $\perp$ phrase is called a partial or incomplete phrase.*

Completable (R0b): *Partial phrases need to be completed during the translation.*

Consistent (R0c): *As translated values can be changed by any other construct, we require that consistency is maintained under these changes. More precisely, the phrase must remain syntactically and semantically valid. Program phrases are complex and may contain internal dependencies. The domain Δ of consistency enforcers δ takes these internal dependencies into account and ensures consistency upon changes.*

4.1.7 Formalization of the Valuation

Defining the Semantics of Language Constructs

4.7. DEFINITION. *The **definition** of language constructs is denoted by a set of semantic definition functions $\mathcal{D}$ which contain the part of their translational semantics which is expressed solely in terms of themselves. These functions ensure the modularity of the construct's translational semantics.*

$\mathcal{D} : (T)^m \times D_1 \times ... \times D_{ex} \rightarrow L_2{}_\perp^\top \times D_1 \times ... \times D_{ex}$ *(m is the maximum arity)*
$\mathcal{T} : D_1 \times ... \times D_{ex} \rightarrow L_2{}_\perp^\top \times D_1 \times ... \times D_{ex}$
$\exists D_i \in \{D_1, ...D_{ex}\} : D_i = L_2$
$\exists D_j \in \{D_1, ...D_{ex}\} : D_j = \Delta$
$\delta : L_2{}_\perp^\top \times D^z \rightarrow L_2$
$\mathcal{D}$ *adheres to requirements R1c, R2b, and R3b .*

R0 - Program representation.

Partial or incomplete values can be constructed using the bottom value $\perp$ which represents completely undetermined information (R0a). A language phrase $F(\alpha_1, ..., \alpha_n)$ with arity n, is called partial or incomplete when $\exists i : \alpha_i = \perp$ or α_i is partial $(0 \leq i \leq n)$

Ppartial phrases can be changed by side effects or mapping state such that they can be completed during the translation (R0b).

Consistency of values can be maintained under the changes (R0c, R1c). The domain Δ of consistency enforcers δ ensure consistency upon integration of an element of an arbitrary domain D^z in a target language phrase. A consistency enforcer is an additional result, next to the target program phrase.

R1 - Compositionality

$\mathcal{D}$ functions denote the translational semantics of source language constructs by consuming an m-tuple of target language constructs and uses these to construct a new target language construct that is the semantic equivalent of a particular source language construct. The m-tuple of target language constructs is obtained by using the $\mathcal{T}$ functions. These functions compute the translational semantics of the parts of a source language construct. Hence, $\mathcal{D}$ functions are compositional and allow us to define the translational semantics of language constructs in relative isolation from other language constructs.

As the translational semantics of a construct does not only depend on the translational semantics of its parts, for each source language construct F a function $\mathcal{D}_F$ is defined so that depending on the source language construct the translational semantics of its parts can be combined.

$\mathcal{T}$ functions prohibit direct access to the terms of the construct and thus ensure R1b. They also support R1a because they can be designed such that they only yield those values which can be consumed by $\mathcal{D}$ functions.

R2 - Multiple Inputs

In order to compensate for the lack of information available in language constructs $\mathcal{D}$ functions also consume an ex tuple of additional information. As such, $\mathcal{D}$ functions can consume the necessary information to be able to define the translational semantics of language constructs (R2a).

$\mathcal{T}$ functions prohibit direct access to the terms of the construct and thus prohibits the access of external information (R2d). However, they do allow the $\mathcal{D}$ functions to control the recursive application of the evaluation. As such, the semantic definition of constructs can affect the context information (R2c). $\mathcal{T}$ functions not only return the translational semantics of a part of

a construct, but also a tuple of external information. As such, $\mathcal{D}$ functions can control the information flow during the recursive evaluation.

R3 - Multiple Results

$\mathcal{D}$ functions also produce an ex tuple of additional information such that the translational semantics of a language construct can produce multiple results which need to be integrated in the translational semantics produced by other $\mathcal{D}$ functions (R3a).

Effecting the Semantics of Language Constructs

4.8. DEFINITION. *The translational semantics of a language construct which could not be expressed in its definition is called the* ***effect*** *and is denoted by a semantic effect function $\mathcal{E}$. It is the part of the translational semantics of a language construct which involves other constructs.*

$$\mathcal{E} : L_1 \times \mathcal{D} \times D_1 ... D_{ex} \rightarrow L_2{}^{\top}_{\perp} \times D_1 ... D_{ex}$$

$\mathcal{E}$ effects the translational semantics of a language construct with the translational semantics of other language constructs by controlling the application of semantic definition functions $\mathcal{D}$ on language constructs.

Upon the application a $\mathcal{D}$ function, the required target program parts necessary to compute the translational semantics of a language construct with the proper additional context information can intercepted in the $\mathcal{T}$ functions. As such, $\mathcal{E}$ can resolve compositionality issues between what is expected by $\mathcal{D}$ functions and what is produced by the valuation function (R1 - compositionality). Also prior to the application of the $\mathcal{D}$ function, the necessary context information can be computed to compensate for the lack of information residing the language construct (R2 - multiple inputs). Anterior to the application of the $\mathcal{D}$ function, the obtained target language construct(s) can be integrated with other already produced constructs (R3 - multiple outputs). Also anterior to the application of the $\mathcal{D}$ function, information necessary for the valuation of other constructs can be prepared. Lastly, compositionality problems can also be anticipated and resolved immediately.

Modular Valuation Functions

4.9. DEFINITION. *A valuation function C modularizes the translational semantics of a language into its language constructs if:*

$$(1) \quad C(F(a_1, ...a_n)) = \mathcal{V}(F(a_1, ...a_n), \overline{d})$$
$$(2) \quad \mathcal{V} : L_1 \times D_1 \times ... \times D_{ex} \rightarrow L_2{}_{\perp}^{\top} \times D_1 \times ... \times D_{ex}$$
$$(3) \quad \mathcal{V}(F(a_1, ...a_n), \overline{x}) = \mathcal{E}(F(a_1, ...a_n), \mathcal{D}_{\mathcal{F}}, \overline{x}))$$

with

$$(4) \quad \overline{x}, \overline{d} \in D_1 \times ... \times D_{ex}$$
(5) $\overline{d}$ *denotes the initial state of the additional information*

A valuation function C that modularizes the translational semantics is defined in terms of a compositional multiple in and output valuation function $\mathcal{V}$ and an initial additional information $\overline{d}$. In this function we combine the *definition*s of the language constructs and their *effect*s.

4.1.8 Higher-Order Grammar Requirement (R4)

The grammar of a language is defined by tree sets containing elements that define the set of well-formed sentences. The set T and N define the vocabulary of symbols which can be used to construct phrases of the language. The set of productions P contain rules defining the set of well formed phrases for the language. With this definition languages are considered as a monolithic entity consisting of symbols and rules.

In iterative language development, a small language is grown in the consecutive phases. This growth is characterized by the vocabulary of the language and its translational semantics. In this section, we focus on the vocabulary part of a language.

Changing the vocabulary of the language means changing the set of productions of the grammar using the vocabulary part. As the productions directly reference each other, the changes required to the rule set are not confined in a subset of the grammar. In order to confine the syntactical definition of a subset of the grammar, we impose the following requirement to context-sensitive grammar definitions:

4.5. REQUIREMENT. ***(R4)*** *The grammar* $\bar{G} = \langle T, N, S, U, D \rangle$ *consists of:*

T *is a set of terminal symbols* ,
N *is a set of nonterminal symbols (syntactic categories)* ,
S *is an element of* N *called start (or initial) symbol* ,
U*are the unbound variables*,
D *is a set of higher order productions, i.e. rules of the form* $K ::= \beta$,
$K \in N$,
$\beta \in (U \cup T \cup N)*$,

Compared to a Chomsky type-2 grammar, the grammar $\bar{G}$ is defined by a set of higher order productions D. The higher order productions differ from the Chomsky production in that they do not always directly refer to other productions, but contain a number of free variables. A grammar complement is used to obtain a Chomsky type-2 grammar.

$\bar{G}$ grammars have a set of unbound variables U. These serve as hooks which are used to compose grammars.

4.10. DEFINITION. *A grammar complement* $\Gamma = \langle \bar{G}_1, ..., \bar{G}_q, B \rangle$ *consists of:*

$B : (D_1 \cup ... \cup D_q) \times P$ *grammar generator* ,
P *is a set of productions, i.e. rules of the form* $A ::= \alpha$,
$A \in N$,
$\alpha \in (N \cup T)*$,
$N \subseteq N_1 \cup ... \cup N_q$,
$T = T_1 \cup ... \cup T_q$,
$S \in (N \cup T)$,

The generator B *binds the variables from* $U_1, ..., U_q$ *to the actual symbols of the grammar* $(T \cup N)$ *in order to obtain the actual chomsky productions of the grammar.*

Example Consider a `select` statement in SQL [CAE+76]. The definition of this statement and all other relevant statements used in this example can be found

in Section 5.1. The grammar of a `select` statement $\overline{\text{Select}} = \langle T, N, S, U, D \rangle$, where:

T = {SELECT, FROM, WHERE},
N = {select, valuelist, sourcelist},
S = select,
U = {Value, Source, Condition},
D = {`select ::= SELECT valuelist FROM sourcelist WHERE condition ,`
`valuelist := Value | Value "," valuelist ,`
`sourcelist := Source | Source "," sourcelist`
}

Complementing the grammar $\overline{\text{Select}}$ with similarly defined grammars $\overline{\text{Expression}}$, $\overline{\text{Column}}$ and $\overline{\text{Table}}$ is defined by the tuple $\langle \overline{\text{Select}}, \overline{\text{Expression}}, \overline{\text{Column}}, \overline{\text{Table}}, B \rangle$ where:

B = {(`select ::= SELECT valuelist FROM sourcelist WHERE condition` ,
`select ::= SELECT valuelist FROM sourcelist WHERE expression`),
(`valuelist := Value | Value "," valuelist` ,
`valuelist := column | column "," valuelist`),
(`sourcelist := Source | Source "," sourcelist` ,
`sourcelist := table | table "," sourcelist`),
}
S = `select`

We do not foresee major difficulties in imposing the above requirement on type-0 or type-1 grammars. We believe that the same reasoning can be used. A thorough investigation lies outside the scope of this dissertation.

4.1.9 Conclusion

We imposed five requirements for modularizing the syntax and the translational semantics of a language construct from other language constructs.

- The first requirement enforces that valuation functions can produce partial program phrases. These phrases need to be completable. The consistency of changes is controlled by local consistency enforcers.
- The second requirement enforces that valuation function operate on language constructs rather than on whole language phrases. We applied the notion of compositionality in denotational semantics. Compositionality is refined in order to cope with the more complex translational semantics of expressive language constructs and enforcing that their valuation does not meddle with the semantics obtained from its parts.

- The third requirement additionally enforces that valuation functions only depend on essential externally provided information. The external information that codetermines the meaning of a language construct should be inferred information and may only be provided by the translational semantics of another language construct if it is essential to define its translational semantics.

- The fourth requirement enforces that valuation functions can define effects on language phrases produced by the translational semantics of other language constructs. The effect can in general only be consumed by other constructs if these effects are essential for their translational semantics.

- The fifth requirement enforces that the syntax of a language construct does not refer to other language constructs.

4.2 Three Language Implementation Concerns

From the above model we deduce a new language implementation design in which languages consists of three kinds of concerns: *basic language concerns* defining the language constructs correspond to the semantic definition functions $\mathcal{D}$, *language specifications* defining the interactions between basic concerns by using *special-purpose concerns* which in turn define the mechanisms to implement interactions, correspond to the effect function $\mathcal{E}$.

4.2.1 Basic Concerns

A basic language concern comprises a *modular* language construct which is defined in *isolation* from the rest of the language implementation. It is defined by a syntactical definition and a translational semantics.

The modularization is achieved by imposing a series of requirements R0 through R4 on the specification of the program fragments, the translational semantics and the syntax. These requirements prohibit the presence of any kind of dependency among the basic concerns. Preserving this separation in the case of complex translational semantics is a challenge. As each expressive language construct is defined in a separate basic concern, isolated from other concerns, its semantics consequently involves other concerns.

4.11. Definition. *A modularized basic language concern* $LC = \langle \bar{G}, \mathcal{D} \rangle$ *is defined by a grammar* $\bar{G}$ *and a set of semantic definition functions* $\mathcal{D}$*, given the main valuation function* C*. The definition of valuation functions* C *and functions* $\mathcal{D}$ *are given in Section 4.1.7.*

4.2.2 Special-purpose Concerns

For each of the requirements R1 to R3, a special-purpose concern offering a mechanism to complete the translational semantics of basic concerns is required. Recall that basic concerns only define the syntax and the translational semantics of language constructs. The effect or the interactions required by the translational semantics of a basic concern with another concern are not part of basic concerns as these interactions violate their modularization. To elucidate this, let us revisit our three sets of requirements R1 to R3: compositionality, multiple inputs and multiple results. Note that the first requirement R0 and the fifth requirement R4 is completely handled by the basic concerns.

- The first requirement is compositionality. Compositionality enforces that the translational semantics of a phrase must be determined by its parts (requirement R1a). Because the algebra for expressing the translational semantics is confined to the grammar of the target language, any value produced by the translational semantics must adhere to the grammar. Therefore, the semantic values obtained from the parts used to construct a new value must be appropriate for constructing the new value. Moreover, as the value represents a program we also expect it to be semantically meaningful. Because the language concerns are isolated from one another, one cannot and should not anticipate all possible values of the parts to yield a valid and meaning full program fragment (requirement R1b). Hence, the first kind of special-purpose concern is a mechanism for resolving compositionality conflicts.

- The second requirement enforces that the translational semantics can in addition only depend on essential externally provided information (requirement R2a). The additional information required for the compilation of a term resides with other concerns or may even have to be computed by other concerns (R2d). Because the language concerns are isolated from one another, the translational semantics containing this information cannot be changed to accommodate for the needs of others (requirement R2c). Hence, the second kind of special-purpose concern is mechanism for controlling context information.

- The third requirement enforces that the translational semantics can define effects on language phrases produced by the translational semantics of other language concerns (requirement R3a). The multiple results produced by the translational semantics of one concern need to be handled appropriately by other the concerns such that the produced results get composed in the correct part of the target program. Because the language concerns are isolated from one another, other language concerns cannot be changed to cope with the additional results and how they affect their translational semantics

(requirement R3b). Hence, the third kind of special-purpose concern is a mechanism for handling multiple results.

4.12. DEFINITION. *A special-purpose concern describes a mechanism to establish a particular kind of interaction among concerns.*

A special-purpose concern does not implement the actual interaction between two or more basic language concerns. It only contains a mechanism which is used to implement the actual interaction. The three kinds of special-purpose concerns need to be addressed for each language concern in order to successfully recompose a set of language concerns into a language. We now further detail the three kinds of special-purpose concerns.

The first kind of special-purpose concern resolves compositionality conflicts. Compositionality enforces that the meaning of a phrase gets computed by its parts. A composition conflict arises when a new semantic value cannot be computed using its parts, due to grammatical and semantical constraints. As compositionality is embedded in the semantical function, the only way to resolve these conflicts is by *intervening* in the composition of these language phrases. Therefore, the interventions need to adjust the definitions the language constructs and/or the compositions with other language constructs. Via such interventions, semantic values can be adjusted to meet the necessary grammatical and semantical constraints.

Non-destructive global interventions are for example supported by monads. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required [New]. As such, the recursive application of the compile function can be controlled by a monad. Hence, monads can intervene in the composition of the program phrases without having to change their composition.

An example of a typical composition conflict is demonstrated in the compilation of Tiger in [App98]. Consider the expression `flag := (a>b)`. In Tiger, boolean expression are compiled into a series of conditional jumps and not to a value. This works fine when boolean expression are used in `if` or `while` constructs. However, as there is no value, an assignment cannot be performed. This conflict is resolved by intervening in the composition of the assignment and the boolean expression.

Other examples are explained in Section 5.7, Section 6.4.1 and in Section 7.4.8.

The second kind of special-purpose concern handles multiple inputs. The realization of these kinds of concerns consists of three parts: identification, obtention and provision.

1. *Identify context information.* The special-purpose concerns issue the communication with other concerns to access information outside the boundaries of the basic concerns.
   Symbol tables for example are very effective as the requester of a value simply accesses the symbol table and retrieves a value by means of an agreed upon key. The key abstracts the concrete location and whereabouts of information.

2. *Obtain context information.* Information can be external configuration information for a given translation, can reside in other concerns or can be the result of a computation spanning several concerns. In the first case, the external configuration must be accessed and looked up. In the second case, the concerns containing the information must be identified. In the last case, the information is distributed among several concerns where each concern relies on the information of another concern to compute its information. A coordinated effort is required among the concerns to compute that information.
   There are many kinds of support for each of three cases. Firstly, global information is supported by a global variable in approaches which are close to general purpose languages. Secondly, information residing in other concerns can be obtained by queries e.g. traversals and structure-shy queries. Lastly, information retrieval in attribute grammars is designed as a coordinated effort.

3. *Provide context information.* The language concerns must be designed so as to allow context information to be passed to it and the language concerns must be able to identify that information. Naturally, information must be computed when it is used in the computation of translational semantics.
   Symbol tables are good examples for illustrating the provision of information. They must be maintained by different language constructs and the information must be available when required.

The best known example where context information is necessary is the declaration of variables and the accompanying scoping rules. Upon using a variable the correct declaration needs to be identified, retrieved and provided. This can be performed by computing the declaration as in attribute grammars (see Section 3.5). Another approach is to thread a symbol table (see Section 3.7.3). Symbol tables are interesting as they raise the issue of providing the correct declaration to a variable usage. The tables are maintained whenever a declaration and scoping changes, and is consulted upon variable usage.

Other examples are explained in Section 5.8 and in Section 7.5.9.

The third kind of special-purpose concern handles multiple results. The realization of these kinds of concerns consists of two parts: identification and integration. We distinguish between local and nonlocal results according to the terminology used in Section 2.3.2. The nonlocal results are the program fragments that affect the results produced by other concerns and are scattered over the target program.

Multiple results are very common in graph rewrite rules, as graph rewrite rule can match any subgraph of the graph under transformation. The approach also illustrates that both source and target programs can be used to identify where the nonlocal term must become part of. The integration of the nonlocal boils down to merely changing the node. That such a simple integration is not sufficient is shown by the integrative composable generators approach.

1. *Identify the terms of which nonlocals must become a part.* In order to identify the term, one can either use the information encoded in the target terms themselves, or use the basic concerns which produced it. Each possibility has a number of distinguishing advantages. The first possibility is to identify the term by using the basic concerns. The advantage of using the basic concerns lies in the level of abstraction provided by them. The identification can be formulated without depending on the details of the valuation of basic concerns. The second possibility is to identify the term by using the target language program. The advantage of using the target program lies in its independence of any particular language concern. In other words, identification depends thus solely on the availability of a target program term, not the concern that created it. When the two possibilities are available in the same system, a hybrid form combining the two possibilities can be used.

   As the term, in which nonlocal terms must be integrated, is an element of the target language, the term is only available after the translational semantics of the producing basic language concern has been executed. Therefore, scheduling is of the essence when locating that term.

2. *Integrate nonlocals with the identified terms.* The integration of the nonlocal must produce a grammatically and semantically correct target language fragment. It must therefore respect the correctness of the nonlocal and the identified term.

Most examples of expressive language constructs (see Section 2.1.1) require changes to be made to the translational semantics of other language constructs. Consider for example a language to produce a traversable tree implementation (cfr. [Bri05]). It consists of language constructs that implement the tree implementation structure and a language construct that produces the traversal methods which need to be integrated in the tree implementation. First, one must

identify the exact location where the methods need to be integrated in the tree implementation. Second, upon integration some dependencies between the generated code of the various language constructs need to be established for an overall correct implementation, such as methods or variables produced by one language constructs that need to be referenced, or used, by the code produced by another language construct.

Other examples are explained in Section 5.9, Section 5.9.2, Section 7.4.10 and in Section 7.5.10.

4.2.3 Language Specification Concerns

We have defined the basic language concerns which define the syntax and the translational semantics of language constructs. We also have identified three kinds of special-purpose concerns which define mechanisms for establishing the interactions among concerns. What remains is the composition of the basic concerns and the special-purpose concerns in order to construct a language. This composition is handled by the last concern, the language specification.

A language specification has two responsibilities: composing the grammar and composing the translational semantics of the language concerns. In order to simplify the definition we use abbreviations like U_{LC} to denote a set U of the definition of a language construct LC.

4.13. DEFINITION. *A language specification LS is defined by the tuple* $\langle S, M, B, C \rangle$

where:

(1) S *is the root* LC,
(2) M *is a set of* LC*s*,
(3) $B : D \times P$ *is a grammar generator*,
(4) C *is a valuation function (see Definition 4.9)*
where
(5) $\forall LC \in M, \exists d$ *in* $D_{LC} : LC ::= \alpha$, *where* $\alpha \in U^*_{LC}$
(6) P *is the set of glue productions*

1. S *is the root language construct of the language. It is the starting symbol of the grammar.*
2. M *is the set of language constructs used in the language.*
3. B *is the grammar generator which composes the language constructs together into a language. The grammar generator composes the higher order productions* D *to form the actual productions* P*. We refer to these productions as the glue productions because they glue the grammars of the various*

language constructs together. The set of higher-order productions D contain a higher order production for each language construct to be used in the language (5). The variables of these productions are the unbound variables of the language constructs U_{LC}.

A language construct can have multiple occurrences within the language, each occurrence composed with various language constructs. Therefore the grammar generator is not a function but a relationship where each higher order production can have multiple bindings, resulting in various productions a.k.a. bindings.

4. *C is the valuation function for the entire language. The function calls the specific valuation functions $\mathcal{D}_{LC}$ for each language construct. The valuation function C acts as a central point of control to guide the evaluation process and manages the interactions between the language constructs.*

The first responsibility of a language specification is to compose the grammar. It is defined by the language construct S and the grammar generator B. From a language specification $\text{LS} = \langle S, M, B, C \rangle$ we can compose a chomsky grammar G. The starting symbol of the grammar corresponds to the starting symbol of the grammar of the starting language construct. The set of terminals contain the language constructs without any unbound variables (the number of unbound variables is denoted by /0). The remaining language constructs are the nonterminals. The set of productions of the new grammar includes the set of actual productions of each LC and the set of glue productions. Formally:

$$
\begin{aligned}
& G = \langle T, N, S, P \rangle \\
\textit{where} \\
& T = \{LC/0 \mid LC \in M\} \\
& N = M \backslash T \\
& S_G = S_{S_{LC}} \\
& P = \{p \mid \exists LC \in M : p \in P_{LC}\} \cup result(B)
\end{aligned}
$$

Note that the translational semantics of the language C is a function that is defined in the language specification LS, it is only used by the language constructs to recursively obtain the translational semantics of its parts.

As the translational semantics of a language construct cannot fully effect its translational semantics due to the set of requirements, the language specification complements the semantical specification of a language. This is the second responsibility of the language specification.

The language specifications rely on the three kinds of special-purpose concerns to establish the interactions that were prohibited by the three requirements

imposed by our modularization model. Each concrete special-purpose concern provides a specific mechanism to implement the interactions. As languages and the necessary interactions vary, so do the special-purpose concerns. In the language specification, a suitable set of special-purpose concerns are selected and applied to establish the interactions among the basic language concerns.

In our formal description, the complement of the translational semantics and the grammar of the basic language concerns is captured in the definition of the valuation function $\mathcal{E}$ and in the grammar generator B respectively. The complements are realized by altering the valuation function $\mathcal{E}$. Recall that this function controls the successive application of the translational semantics of the language constructs $\mathcal{D}_F$. For each language construct, control can be exercised over the parts, over the external information and over the produced results. As such, compositionality conflicts can be resolved, external information flow can be set-up and multiple results can handled correctly. The complements of the basic language concerns are thus externally defined extensions to the language constructs. As such, the entire translational semantics of a language construct has a local footprint situated around the language construct itself. This locality respects the dominant decomposition of a language into language constructs, and preserves the separation of concerns we set out in the beginning of this section.

4.3 Separating Special-purpose Concerns

The challenge we face in this dissertation is to *capture the three kinds of special-purpose concerns discretely in separate modules* without breaking the separation of the basic language concerns.

The goal of special-purpose concerns is to ensure that the interactions do not require any invasive alteration of the basic concerns (requirement R1, R2, R3, R4). Separating the special-purpose concerns from the basic concern is a delegate process. The modularization of a special-purpose concern must respect the dominant decomposition of a language into basic concerns. *Semantics that is specific to a particular interaction must remain local to the special-purpose concern. Semantics that is specific to a particular basic concern must remain local to that basic concern.*

We distinguish between two kinds of translational semantics that are added to the basic language concerns: translational semantics complements and concern-specific logic.

4.14. Definition. *(SP0a) Translational complements effect the translational semantics of basic language concerns.*

4.15. Definition. *(SP0b) Concern-specific logic is semantics that does not contribute to the translational semantics. It is semantics on which other language*

concerns are relying as required by the necessary interactions between the concerns.

Let us revisit the special-purpose concerns and discuss the design challenges raised by the three special-purpose concerns. It is not our intension to present an exhaustive list of design challenges. This is in our opinion not possible. The list we compiled is based on our extensive study of contemporary LDTs. So each challenge has been addressed to a certain extent by these LDTs. This becomes clear when we evaluate, in Section 4.5, these LDTs against the identified challenges. Note that all challenges are illustrated by an example in Sections 5.5 to 5.9.

4.3.1 Challenges to Separate the Resolution of Compositionality Conflicts

Compositionality conflicts can be resolved by interventions in the composition of the language. The interventions change how a basic concern is composed with other concerns. There are two challenges to resolve compositionality conflicts.

SP1 - Localized interventions One of the difficulties is to keep this change *local* to the basic concern, because changes in composition may invalidate translational semantics complements based on other special-purpose concerns e.g. information obtention. An additional example of localized intervention is detailed in Section 6.4.1 and in Section 7.4.8.

SP2 - Global interventions Another challenge is to solve composition problems which are not specific to a specific basic concern, but have a *global* impact on a larger set of interactions between basic concerns.

4.3.2 Challenges to Separate the Handling of Multiple Inputs.

SP3 - Identification with abstract names Context information is derived from the source language (requirement R2b). The intention behind this information serves as an identification mechanism in order to abstract from the location of that information and in order not to pollute a basic concern with the semantics of other basic concerns. In other words, the intentions enable communication between the basic concerns without erecting dependencies between them. Moreover, the computation of derived information which is specific to a concern must remain local to that concern, in order to respect the separation of a language into basic concerns.

Obtention We distinguish between three sources of information, as each poses its own challenge. An additional example illustrating SP5 is given in Section 7.5.9.

SP4 - Obtention of External Information This is the most simple form of information obtention. It requires a mechanism to obtain information outside of the language definition.

SP5 - Obtention of Information of another language concern The challenge to obtain the information of another concern is the organization of the flow between the language concern that requests the information and the concerns that possess or influence the information. The definition of the basic language concerns must not be altered in order to compute the necessary information (requirement R2c). The challenge lies thus in separating the flow of information of the basic concerns and minimizing the dependencies with the basic language concerns. Note that, a concrete obtention of information specific to a concern should remain local to that concern. Only the mechanism for the obtention is separated into a special-purpose concern.

SP6 - Obtention of information, distributed among several concerns There is an additional challenge in the case of distributed information. In the retrieval of distributed information, every involved concern performs a part of the computation that is specific to that concern. These computations are usually hierarchically organized, i.e. the computation depends on the results of other participating concerns. While we stress that separation, concerns should remain the dominant decomposition in the language. So in case the information is distributed among terms, the logic that is dependent on basic concerns may not be part of the special-purpose concern. Hence, concerns should be externally modifiable or extendable with additional logic once we use a concern in a particular language.

SP7 - Provision Offering the required information to the language concerns should respect the modularization of the concern. This means that the information has to be provided at the time it is requested by the concern without changing its definition. The evaluation process must ensure that computation of the information is scheduled before it has to be provided.

4.3.3 Challenges to Separate the Handling of Multiple Results

Identification The term in which to integrate the nonlocals is part of the target program. We distinguish between two possibilities for identifying this term: either identify the term via the source language program or identify the term via the

target language program. Each possibility has different tradeoffs and challenges to overcome. In Section 5.9.2 the tradeoff is discussed more in detail and is illustrated with an example. In that particular example, the target program is the best choice. Another example illustrating that identification via the source program is a better choice is discussed in Section 7.5.10.

SP8 - Identify via the source language program The first possibility to identify the term is to use the input source language program. One should be cautious not to violate the separation of concerns, because of several reasons: First, the concerns producing the term in which a nonlocal ought to be integrated, cannot be charged with additional responsibilities in order to help identify or even actively search for nonlocals. Second, finding the proper language concern may prove to be challenging. As each semantical value is used by another language concern which recomposes with other values, the ultimate role of the semantical value in the target program may change during evaluation. It is often the role which codetermines in a target program fragment whether a nonlocal should be integrated or not. Hence, relying on the language concerns to determine the role a semantical value fulfills in the target program may require detailed knowledge about all the language concerns influencing the function of that semantical value. Such dependencies penetrate the separation of the basic concerns.

SP9 - Identify via the target language program Another possibility is to steer the identification based on the target language program. As the target program is produced by various basic concerns, identifying the term erects dependencies between the semantical values produced by all the basic concerns along the path between the nonlocal and the term in which to integrate the nonlocal. Due to the difference in abstraction level between the source and the target language, the semantical values are more verbose. As the intention (denotation) of the language constructs is spread over a larger program fragment consisting of generic constructs, the identification of the correct term requires more details about the semantic values. As such, the increased risk of having more detailed dependencies compromise the separation of concerns and renders the identification process more fragile. This risk should be minimized as much as possible.

SP10 - Scheduling The last issue complicating the separation of this special-purpose concern is scheduling. Basic concerns should be free to produce any semantic value regardless of the fact whether other concerns producing the identified term or influencing the integration have already been executed or not.

Integration The integration of a nonlocal must produce a grammatically and syntactically correct target language fragment. In its simplest form, integration

is a three-party contract. More complex integrations are context dependent. A well-balanced implementation of these challenges, that preserve the separation of concerns is discussed in Section 6.3.3.

SP11 - A three-party contract The concerns producing the term in which a nonlocal ought to be integrated, cannot be charged with additional responsibilities to integrate the nonlocals. A basic language concern must be separated from the integration mechanisms. However, the basic language concerns cannot be completely oblivious to integration. To this end, a composition is a third-party contract consisting of the two program fragments that need to be composed and an external actor encoding that composition. The parties which are being composed must be actively involved in this process, in other words the program fragments may not be treated as mere data and serve as input for the external rules. Otherwise, the integration itself must remain external to the concerns.

SP12 - Context-dependent integration More complex integrations depend on the enclosing terms. Successive applications of the valuation function, recompose a semantical value with other values. The enclosing terms determine how semantical values are used within a target program. As the nonlocal result is just one of the results of the semantical valuation functions, the nonlocals are also subject to these successive recompositions. Therefore, the integration of the nonlocal result may also be influenced by these enclosing terms. We call such integrations context-dependent integrations. The dependencies with the enclosing terms cannot be avoided, but should be captured separately from the basic concerns that produced them.

4.4 Evaluation of the Separation of Concerns

We evaluate the formalization of the separation of language constructs in our new language implementation design which consists of basic concerns and various kinds of interactions a.k.a. special-purpose concerns.

Each concern necessary to implement a language has been made explicit: basic concerns, special-purpose concerns and language specifications glueing these together.

The separation into basic and special-purpose concerns is general and succinct. The basic and special-purpose concerns contain the stable parts of a language implementation, respectively a single language construct and a mechanism to establish a particular interaction among the former. Each basic concern defines the translational semantics of a language construct irrespective of the language in which this basic concerns will be used. Each special-purpose concern is designed to capture a particular interaction pattern among the basic concerns. The stable and repetitive tasks of interactions are defined in the special-purpose concern.

The variable part is implemented via the special-purpose concerns in the language specification.

Basic language constructs can be more easily certified because their definition and their translational semantics can be studied in isolation. No other logic and dependencies are tangled in the concerns that obstruct and complicate reasoning. The same is true for special-purpose concerns. Each mechanism to establish a particular interaction can be subject to extensive testing. Furthermore, special-purpose concerns can access the concrete language specification and check some preconditions on a concrete language upon compilation of the language specification.

We evaluate the separation of concerns of a language implementation using the following criteria:

Not tightly related The basic concerns are defined in complete isolation from one another to the extent that their grammatical and semantical definitions are restricted. In their respect, these concerns are completely orthogonal. Introducing them into a language does not affect the translational semantics of other concerns. However, the language specification complements the translational semantics of these concern with logic that enables the necessary interactions among these concerns. These interactions render the basic concerns not orthogonal in their composition. The degree to which this logic is robust to changes depends on whether changes to the language break the interaction defined by the mechanisms provided by special-purpose concerns.

Cohesion At first sight, basic concerns may not fully adhere to this principle. Basic concerns only define their translational semantics and have a higher-order grammar. In other words, basic concerns only have a partial grammatical and partial semantical definition. However, when basic concerns are used in a language (see Section 4.2.3), they are further completed with semantic complements (see Definition 4.14) and concern-specific logic (see Definition 4.15). As such, the complete translational semantics of a language construct logically become part of the basic concern that defines it. Changes or modifications specific to a basic concern or its interactions with other concerns can be performed in the same logical entity. (cfr. the challenges of the special-purpose concerns in Section 4.3 and the language specification in Section 4.2.3).

The complexity of separating concerns. Ernst et.al. [Ern03] formalizes the concept of separation of concerns by means of a unary homomorphism. Assume that we have a programming language L. Let P be the set of all programs written in this language. Moreover, let S be the set of all possible semantic values for L, such that every program $p \in P$ has a semantics $s \in S$. Finally, let $\phi : P \longrightarrow S$

be the semantic function, such that $\phi(p)$ is the semantics of the program p. Given P , S , and ϕ, and functions $\pi_{syn} : P \longrightarrow P$ and $\pi_{sem} : S \longrightarrow S$

- ϕ is a unary homomorphism from (P, π_{syn}) to (S, π_{sem}) if $\forall p \in P$: $\phi(\pi_{syn}(p)) = \pi_{sem}(\phi(p))$.
- $\pi_{syn} : P \longrightarrow P$ is a syntactic reduction function which removes the concern from the program. Similarly, $\pi_{sem} : S \longrightarrow S$ is a semantic reduction function removing the concern from the meaning of the program.
- $\phi(\pi_{syn}(x)) = \pi_{sem}(\phi(x))$ ensures consistency between the syntactic and the semantic level. In other words, the syntactic reduction function is used as a tool to specify exactly what the concern looks like, and the semantic reduction function is used to specify exactly how the concern works.

Ernst et.al. defines separation as a property of a reduction function, i.e., a concern may be either syntactically or semantically separated from the rest of the program. This analysis framework aids in deciding on how much separation is implied by a given set of reduction functions. A syntactic reduction function exhibits a high degree of separation if it is simple e.g. it deletes a single subtree from the program. A semantic reduction function exhibits a high degree of separation if it is simple i.e. has no other effect than *skipping* the initialization of one or more variables (global/static variables, or instance variables) as well as some operations on them.

The semantic reduction function to remove a basic concerns from language is relatively simple given the fact that concerns are not orthogonal in their composition. The simplicity is mainly due to the use and the separation of the interaction mechanisms into special-purpose concerns. The use of interaction mechanisms guarantees a certain degree of robustness to changes which do not break the interaction mechanisms. Because the interaction mechanisms are separated, the complex logic of the interaction mechanism does not have to be considered in detail when removing a basic concern from a language.

In order to remove a particular language construct LC from a language specification $LS = \langle S, M, B, C \rangle$, the language specification is altered into a new specification LS' such that:

- The new set of language constructs M' no longer contains the language construct LC
- The grammar generator B rebinds the abstract productions D that were bound to the language construct LC to another language construct.
- The new valuation function C':
  - fixes the information flow which is broken due to the structural change or because an information source is missing.

- fixes the distribution of nonlocal values which are broken due to structural change or because the semantic value to integrate the nonlocal value is missing.
- fixes the compositionality conflicts due to the new language construct compositions B_{new}

$$
\begin{aligned}
& LS' = \langle S, M', B', C' \rangle \\
& \textit{where} \\
& M' = M \backslash \{LC\} \\
& B' = B_{unchanged} \cup B_{new} \\
& B_{unchanged} = \{(d, p(a_1, ..., a_n)) | a_i \in M'\} \\
& B_{new} = \{(d, p(a'_1, ..., a'_n)) | a'_i \in M' \\
& \qquad \text{and } \exists (d, p(b_1, ..., b_n)) \in B \textit{ with } b_i = LC\} \\
& C' \text{ is the new valuation function}
\end{aligned}
$$

The syntactical reduction function is similar to the semantical reduction function due to the cohesion of the basic language concerns in the language specification. Naturally, a concrete implementation must ensure this property.

4.5 Interaction Strategies

Based on our analysis of the contemporary language development techniques (LDTs) (see Definition 2.2) presented in Section 4.5.1, we find that the LDTs lack the ability to effectively modularize the basic language concerns and the special-purpose concerns. However, our evaluation revealed a number of mechanisms which indicate that LDTs offer special mechanisms to implement a particular task or challenge of a special-purpose concern. We call these mechanisms *interaction strategies* (see Section 4.5.2). In a thorough evaluation performed in Section 4.5.3 and Section 4.5.4 of these mechanisms, we explore four interaction strategy shortcomings. From these shortcomings we distill in Section 4.5.5 the necessary design requirements for a new language development technique in which these shortcomings can be tackled.

4.5.1 Separation of Concerns in Contemporary Language Development Techniques

Each contemporary language development technique provides its own abstractions and techniques which can be used to implement languages. In this section, we present a summary of the investigation of the degree to which the LDTs adhere

to or are capable of keeping the identified concerns separate. The full analysis is described in Appendix A.

The investigation of each concern in a particular language development technique is structured according to the major tasks and major challenges identified in the previous section. There are two questions we answer in the discussion of each task and challenge. First, is it possible to separate the various concerns. Second, to what extent are the mechanisms offered by the systems sufficient.

An overview of the analysis is shown in Table 4.3. The table lists the various kinds of LDTs which are presented in Chapter 3. We discuss them against the task and challenges required for the effective separation basic concerns and special-purpose concerns. The task and challenges are the columns. The list of abbreviations used in those columns are explained in Table 4.2. The various entries of the overview table are listed in Table 4.1.

Observe that some LDTs are discussed as a group (such as template-based approaches) if they do not have distinct properties which are relevant to the goal of this investigation. Other groups of systems (such as ad hoc systems or compositional generators) are discussed in more detail for exactly the opposite reason.

Symbol	Explication
-	The task or challenge is not applicable or cannot be defined for that system, because the entry represents a heterogeneous group
	The entries in the table which are left blank indicate that the task or challenge cannot be realized.
·	Indicates that a solution is possible but in practice not feasible.
○	Indicates that a solution is possible, however it is not or ill supported by the LDT.
●	Indicates that the solution supports a degree of separation of concerns.
⋆	Indicates that a LDT offers special mechanism to implement a particular task or challenge of a special-purpose concern.

Table 4.1: The different degrees of separation of concerns

Table 4.3 contains varying degrees of separation of concerns offered by the contemporary LDTs. The results of our investigation are promising but unsatisfactory. The promising part is that there is always a LDT which has a ● for each column of the table. This means that for each requirement, task and challenge there exists a LDT which offers at least a degree of separation of concerns. Most columns even are marked with a ⋆, which indicate that an LDT offers special support. Hence, it is safe to say that each of requirement, task and challenge plays an important role in the development of language implementations.

The unsatisfactory result of this investigation is that there is no row which is filled with ○, ● or ⋆'s. In other words, there is no LDT in which all of the requirements of the basic concerns and of the special-purpose concerns can be

basic concern	
GS	grammar and semantics
Ch	completable
P	partial
Co	consistency
MI	multiple inputs
MO	multiple outputs
Special-purpose Concern - Compositionality	
LI	localized interventions
GI	globalized interventions
Special-purpose Concern - Multiple Inputs	
Id	Identification
Obe	Obtention of External Information
Obc	Obtention of Information of another language concern
Obd	Obtention of Distributed information
Pr	Provision of Information
Special-purpose Concern - Multiple Results	
IdS	Identify via the target language program
IdT	Identify via the source language program
S	Scheduling
I3	Integration using a three-party contract

Table 4.2: Tasks and challenges of language concerns.

implemented, let alone supports a degree of separation of concerns. We can thus conclude that a language implementation in each of the listed LDTs is bound to violate the separation of basic and special-purpose concerns.

There are entries that deserve a more thorough discussion. Those are marked with a $\star$. A $\star$ indicates that a LDT offers a special mechanism for implementing a particular task or challenge of a special-purpose concern.

4.5.2 A Definition of Interaction Strategies

The interesting thing about the entries marked with a $\star$ is that these are succinct, distinct, successful and proven approach to implement the special-purpose concerns that confine the impact of the implementation of complex translational semantics on other basic concerns. We will call these mechanisms *interaction strategies*.

Interaction strategies do not entirely separate the various concerns within a language implementation. The most noteworthy interaction strategies in this regard are those that deal with the integration of multiple results as a three-party contract. Monads, ICG, SOP and dynamic rewrite rules are four examples in which the weight of the integration is focused on: external rules, the concerns that consume the nonlocals, the target language translational semantics and the concerns that produce the the nonlocals (this is discussed in more detail in 4.5.4). Clearly none of them actually support the three-party contract (SP11 4.3.3) as each interaction strategy shifts the responsibility of integration to one entity. As such, these interaction strategies violate separation of concerns.

4.5.3 Interaction Strategy Space

The most prominent interaction strategies found in our analysis are listed in Table 4.4. The table groups the interaction strategies into six major families of systems: tree rewrite rules, graph rewrite rules, attribute grammars, template-based approaches, compositional systems and ad-hoc systems. Each interaction strategy is designed so as to improve the separation of a kind of special-purpose concern.

When looking at Table 4.4 three things catch our immediate attention. First, interaction strategies are *unequally spread* over the various tasks of special-purpose concerns. Obtention of information for multiple input concerns is strongly represented in the table, whereas interaction strategies for multiple results, and compositionality are rather rare.

Second, there are no *all-in one* interaction strategies. Interaction strategies focus on a particular task of the concern. So for instance in order to control multiple inputs, several interaction strategies need to combined. Another less trivial example illustrating the need to combine multiple interaction strategies is for localizing compositionality interventions by means of interaction strategies for

the obtention of external information. By implementing the obtention of external information with interaction strategies that can cope with the changes induced by the interventions, interventions cannot break the obtention of information.

Third, there are very few interaction strategies which *reoccur* in multiple LDTs. You can see this rather quickly by looking at the name of the interaction strategies. Names are strongly related to the LDT which support that interaction strategy. There have been attempts with limited success, for example the construction structure-shy query facilities with rewrite strategies [vW03]. The only systems where interaction strategies often reoccur are ad-hoc approaches. The same cross-fertilization between general purpose languages and LDTs occurred in attribute grammars: a favored platform for constructing attribute grammars is functional and object-oriented languages [Paa95].

4.5.4 Interaction Strategy Shortcomings

Based on our three observations above, the current interaction strategy space can be characterized as incomplete, fragmented and non-orthogonal to the LDTs. The space is incomplete due to their unequal spreading over the tasks of the special-purpose concerns. The space is fragmented because there is no all-in one interaction strategy. And finally, the interaction strategy space and the LDTs is non-orthogonal because interaction strategies seem to be largely specific to a particular LDT.

Given the current configuration of the space, LDTs fail to separate completely the concerns using their interaction strategies. Fragmented interaction strategies only improve the separation of concerns for a particular task. So for a complete separation, different interaction strategies focussing on other tasks and concerns must be combined. Unfortunately the set of interaction strategies is incomplete, so there are interaction strategies lacking. However, completing the interaction strategies is not trivial as interaction strategies cannot be added to the LDTs.

In search of some answers to resolve the above situation, we conducted a closer analysis. The analysis revealed four shortcomings of contemporary interaction strategies: they are not generally applicable, there is room for improvement, there is room for new interaction strategies and there is no silver bullet interaction strategy.

Interaction Strategy applicability

Each LDT offers its proper interaction strategy with its strengths and weaknesses. For each challenging language implementation a choice must be made in order to determine the most applicable interaction strategy. Each interaction strategy captures a particular interaction pattern between concerns. So, the suitability of a particular interaction strategy depends on whether the interaction pattern captured by the interaction strategy conforms to the communication pattern among

basic concerns.

In Appendix B, we compare the interaction strategies that improve the separation between interacting basic concerns by focussing on the same task or challenge of the same special-purpose concern. We show that interaction strategies implementing the same task or challenge of a special-purpose concerns have different tradeoffs and are not simply exchangeable.

Room for Improvement

There is still room for alternative interaction strategies to improve current interaction strategies. An obvious example are monads. Monadic programming is a general purpose feature which offers control over the application of the translational semantics of the basic concerns. Its generic property opens many opportunities, but its bareness renders it a crude tool for separating the concerns of a language implementation. Clearly, monads were not designed with this purpose alone in mind. However, monads teach us that there is large potential to be explored.

Current interaction strategies are only capable of establishing a degree of separation of concerns, hence our rather lengthy discussion. Interaction strategies have a number of inherent properties but also have a number of limitations due to the properties of the LDT in which they are embedded. An example of such an interaction strategy is attribute forwarding. Attribute forwarding redirects attribute requests to the target language fragment. One of the difficulties of this interaction strategy, which is acknowledged by the authors, is to determine when to redirect and where to direct. Children, parent and results may define the attribute, and the order in which these are consulted is difficult to generalize for every language implementation. Unfortunately, attribute grammars do not expose control over the attribute requests.

Current interaction strategies provide alternatives for explicit referencing other concerns. More so, these interaction strategies target the accidental involvement of concerns. We believe that it is also possible to eliminate some of the more essential involvement of concerns. Consider for example symbol tables and structure-shy queries. In the symbol table interaction strategy, concerns that produce the information that needs to be accessible for other concerns, must still explicitly publish that information into the symbol table. The explicit publication can be extracted from of those concerns in cases where the value that has to be published can be identified by exploiting structural or run-time information. Structure-shy queries reference other concerns directly. This could be avoided by extending structure-shy queries for finding concerns based on other information such as the availability of some derived information or some structural information. Another example is the scope of structure-shy queries. The scope of XPath structure-shy queries is unlimited. A query can traverse the entire source or target program. For some queries that is not desirable and the search space should be limited.

By adding a scope clause to structure-shy queries this can be realized. In Section 6.3.2, we extend our structure-sky query interaction strategy with scoping.

Room for New Interaction Strategies

Interaction Strategies are unequally spread over the various tasks of special-purpose concerns. Most interaction strategies are mainly focused on the propagation of information. New interaction strategies can be conceived to tackle other kinds of involvements such as changing the responsibilities and the implementation of other concerns to ensure cooperation and consistency. The most prominent example of such a situation is the underrepresentation of specific interaction strategies to handle multiple results. In most LDTs multiple results are currently treated as multiple inputs. This means that the identification of concerns in which nonlocals must be integrated is made responsible for retrieving nonlocals and for integrating them in the translational semantics of the concern. Interaction strategies which do support the identification, statically introduce anchor points to guide the identification e.g. dynamic rewrite rules are statically scoped, and morphisms describe static patterns. Statically deciding where to integrate is in general not always feasible (cfr. implicit node creation). These mechanisms are especially unsuited to deal with context-dependent integration. In Section 6.3.3, we implement a new interaction strategy to handle multiple results, which is called INR.

So far interaction strategies are generic, meaning that they can be used in many language implementations. There are also interaction strategies which are designed for a particular language specification. We designed such an interaction strategy in Section 7.5.9 for the case study used to validate this dissertation.

The applicability of interaction strategies is limited because they focus on particular tasks or challenges and assume a different set of prerequisites and interaction patterns between the concerns. Hence, new interaction strategies tailored to other interaction problems and language cases are necessary to separate the concerns of a language implementation effectively. In Section 6.3.3 we present a series of language-specific extensions to the INR interaction strategy that handles multiple results. These extensions complete the basic version of the interaction strategy in order to incorporate the semantics of the language.

Silver Bullet

Each LDT advocates its own, specific interaction strategy(-ies) for improving the SOC of a language implementation. Furthermore, each of them have been researched in depth by the community and have proven over time their value.

Based on the current interaction strategy space, interaction strategies need to be combined so as to provide an adequate solution (e.g. multiple inputs interaction strategy and localizing compositionality resolutions in Section 4.5.3). Moreover,

interaction strategies make different tradeoffs. Consider for example structure-shy queries and attribute propagation rules. In the former, one explicitly and locally defines a path in the parse tree without imposing any requirements on other concerns. In the latter, attribute values are requested (more or less) regardless of location of the value of that attribute but relying on other concerns to offer (define, propagate) the attribute.

It is unlikely that an overall general-purpose interaction strategy can be found that combines all the merits of the existing interaction strategies, eliminates their drawbacks and thus renders them obsolete.

Conclusion

Contemporary interaction strategies alone do not suffice because of the four shortcomings explained in the previous paragraphs. Additionally, we observe that interaction strategies are embedded in the language development techniques. They are offered as part of the distinguishing characteristics of a LDT. The availability of such an interaction strategy determines for those systems to a large extend their competitive advantages over other systems. It is thus desirable to equip a LDT with appropriate facilities in which new and specially tailored interaction strategies can be specified.

4.5.5 Metafacilities

The interaction strategies embedded in contemporary LDTs do not suffice for completely separating the basic concerns from the special-purpose concerns. Therefore a LDT must be equipped with facilities to be able to construct new and tailored interaction strategies. The question we asks ourselves in this section is what kind of facility is suitable. To answer that question, we need to find a common denominator for characterizing interaction strategies in the midst of the design choices of a LDT.

The Representation of Programs

The first design choice of LDTs is the representation of programs. This representation dictates the available operations which can be performed by the transformation modules of a LDT. There is a tension between the various forces influencing that decision.

From a separation of concern perspective, data structures must be completable and basic concerns should enforce the local consistency of the produced program fragments. The only way to enforce local consistency of data structure is to construct an abstract data type.

The simplest way to discover the influences of interaction strategies on the data structures is by looking at reoccurring interaction strategies in ad-hoc approaches. It comes at no surprise that the most reoccurring interaction strategies are found in ad-hoc approaches. The major reason for this is due to the availability of a general purpose language. A general purpose language renders ad-hoc systems the most flexible LDTs, in which new libraries can be added and implemented to facilitate the implementation of languages. General purpose languages provide good libraries to query tree structured data if a suitable representation is used. Having a suitable data structure is thus an important enabler. In most ad-hoc approaches, such a suitable structure is a metastructure (e.g. container - element structure). The metastructure provides a generic interface to access and manipulate the specific data structure of each language construct. As such, it is used to facilitate queries such as traversals in OO, in prolog and in functional languages like Haskell. Some LDTs require developers to use these metastructures directly, some generate this metalevel and others use reflection.

Because the two representations are important they should be available at the same time: an abstract data type to implement the basic concerns and a metastructure to implement and separate the special-purpose concerns.

Amount of Control LDTs Expose over their Execution

The second design choice is the amount of control the LTDs expose over their execution. Each interaction strategy exposes, in a very controlled way, part of the system. So each interaction strategy enables reasoning about the metalevel confined and controlled to a particular purpose. Table 4.5.5 lists all the identified interaction strategies together with the metaoperations they perform on the system. Because the interaction strategies only expose part of the system in a restricted fashion, it is hard to construct other interaction strategies or even adapt them. The attempt to implement structure-shy queries on top of rewrite strategies has been successful up to the point where structure-shy queries required more metalevel abilities.

The focus of the LDTs at the moment when the interaction strategies were designed was to offer a *single* suitable layer of abstraction for the internal machinery of a LDT. Nothing more and nothing less. Those highly restricted metalayers expected that the interaction strategy offered to separate the concerns is sufficient for every language implementation. A typical example illustrating this expectation of the designers of interaction strategies are the long awaited specific interaction strategies to implement language concerns with multiple results. In many LDTs multiple results are treated as the inverse of multiple inputs and should therefore be tackled with interaction strategies to control multiple inputs. Interaction strategies and capabilities to control multiple results have been added very recently. Compared to the first interaction strategies to control the multiple inputs that were designed in the late seventies, the interaction strategies to con-

trol multiple results were published in 2001 (dynamic scoped rewrite rules) and 2005 (ICG).

The critique that reflective systems rely on the assumption that their metafacilities are sufficient for the domain of specific system extensions was first presented in the context of support for evolution by Kai-Uwe et.al. [MB97]. Kai-Uwe even argues that "the developers have to devise an appropriate design for that higher level self-reflection". In this dissertation we argue exactly the same line of thought. Interaction strategies should be designed by developers such that a higher level of reflection can be employed by the language developers. To make this possible LDTs must be equipped with more generic metafacilities. More generic metafacilities do not imply unlimited and full control over an LDT as it is the case with ad-hoc approaches. Neither does generic metafacilities imply a general purpose mechanism (like monads) which merely is the vehicle to implement interactions. Generic metafacilities for interaction strategies must expose an appropriate expressiveness at a suitable abstraction level for controlling the execution and to reason about an LDT and the language implementation at hand. We want to stress with the terms appropriate and suitable that metafacilities of an LDT should be designed for supporting the construction of interaction strategies for resolving compositionality conflicts, control the information flow and integrate nonlocal results.

4.6 Conclusion

Modularization Model We separate, decouple and parameterize the language constructs to modularize a language implementation along the dimension of the language constructs.

Our modularization model is defined by five requirements R0 to R4.

The five requirements are:

- The first requirement (R0) enforces the that valuation functions can produce partial program phrases. The consistency of changes is be controlled by local consistency enforcers.

- The second requirement (R1) enforces that valuation functions operate on language constructs rather than on whole language phrases.

- The third requirement (R2) enforces that valuation functions can in addition only depend on essential externally provided information.

- The fourth requirement (R3) enforces that valuation functions can define effects on language phrases produced by other the translational semantics of other language constructs.

- The fifth requirement (R4) enforces that the grammar of single language constructs are set of higher order productions which do not directly refer to other productions, but contain a number of free variables.

The requirements R0 to R3 imposed on the specification of the translational semantics prohibit the presence of any kind of dependency among the basic constructs, ranging from a direct reference to other constructs, or any implementation decision that is imposed by, or stems from, another construct. The fifth requirement R4 attacks the monolithic definition of grammars of languages in order to separate the grammatical definition of each basic language construct. As such, basic language constructs define their translational semantics and their syntax in complete isolation from one another.

Three Language Implementation Concerns From our modularization model we deduce a new language implementation design in which languages consist of three kinds of concerns: *basic language concerns* defining the language constructs, *language specifications* defining the interactions between the basic concerns by using *special-purpose concerns* which define the mechanisms to implement the interactions.

Each basic concern comprises a modular language construct which is defined in isolation with respect to the rest of the language implementation. It is defined by a syntactical definition and a translational semantics. As such, a basic concern captures a separate construct of a language definition.

The five requirements restrict the definition of the syntax and the translational semantics of basic concerns. The exiled translational semantics needs to be completed, to be able to effect the complex translational semantics. Requirements R1 to R3 are tackled by three kinds of special-purpose concerns. A special-purpose concern describes a mechanism for establishing a particular kind of interaction among concerns:

- The compositionality requirement is completed by the special-purpose concern that resolves compositionality conflicts.
- The requirement enforcing that translational semantics can in addition only depend on essential externally provided information is completed by the special-purpose concern that controls multiple inputs.
- The requirement enforcing that translational semantics can define effects on language phrases produced by the translational semantics of other language concerns is completed by the special-purpose concern that controls multiple inputs.

The challenge lies in describing the mechanisms of these special-purpose concerns separately from the basic language concerns such that the separation of basic language concerns is not violated: invasive changes of basic concerns are prohibited and their cohesion is ensured.

Completing basic language concerns to compensate for the requirement R1 to R4 is performed in a language specification. The specification binds the syntactical definitions of basic concerns and uses special-purpose concerns to establish interactions between them.

Metafacilities for Interaction Strategies An analysis of the separation of concerns into basic concerns and special-purpose concerns in contemporary LDT shows that separation cannot be completely realized. The success of LDTs range from entire failure to implement the given concerns, to at most a couple of mechanisms to improve the separation of concerns. We refer to these mechanisms as interaction strategies.

The current interaction strategy space can be characterized as incomplete, fragmented and non-orthogonal to LDTs. Given the current configuration of the space, LDTs fail to separate basic concerns completely using their interaction strategies. A closer analysis revealed four shortcomings of contemporary interaction strategies: they are not generally applicable, there is room for improvement, there is room for new interaction strategies and there is no silver bullet interaction strategy. In order to remedy this situation, these shortcomings can be tackled by equipping LDTs with metafacilities so as to be able to construct new and tailored interaction strategies.

The metafacilities form a common denominator of interaction strategies which determines the representation of the programs and the amount of control which is exposed by a LDT. Two representations are important and should be available at the same time: an abstract data type to implement the basic concerns and a metastructure to implement and separate the special-purpose concerns. Interaction strategies should be designed by developers such that a higher level of reflection can be employed by language developers. To make this possible, generic metafacilities are required.

Chapter 5 presents a new language development technique and discuss the basic concerns and the language specification concern. Chapter 6 discusses how special-purpose concerns fit in the new language development technique. It introduces the metafacilities required for implementing the tasks and challenges imposed by special-purpose concerns.

LDT/Features	basic concerns						Compositionalty		Multiple Inputs				Multiple Outputs				
	GS	Ch	P	Co	MI	MO	LI	GI	Id	Obe	Obc	Obd	Pr	IdS	IdT	S	I3	Ic
Rewrite Rules		●	○		○	○	○	○		·	○	○			·			○
Traversals		●	○		○	○	○	○		○	⋆	●			●			●
Dynamically scoped rewrite rules with rewrite strategies (focus on Stratego)			●	○	○	⋆	○	○		○	⋆	○			⋆	⋆	⋆	
Macros (focus on Macros designed for Common Lisp)	●	○	○				·			●	●				○	-	-	-
Template-based Approaches (like XSLT, Velocity)		○	○		○	●	·			●	⋆	○				-	-	-
Graph rewrite rules (like AGG)		●	●		○	●	○	○		●	●	○	○	○	○	●	●	
Implicit Node Creation		●	●		○	●	○	○		●	●	○	○	●	⋆		⋆	
Matching by Morphisms		●	●		○	●	○	○		●	⋆	○	○	○	⋆		●	
Higher Order Attribute Grammars					○	●	⋆	·		⋆	●	●	⋆	●		○	●	○
Forwarding					○	●	●	·		⋆	●	●	⋆	⋆		○	●	○
Multiple inheritance and Templates					○	●	●	·		⋆	●	⋆	⋆	●		○	●	○
First Class Attribute Grammars					○	●	●	·		⋆	●	⋆	⋆	●		○	●	○
Compositional Generators	-	-	-	-	-	-	-	-		-	-	-	-	-	-	-	-	-
Subject oriented programming (SOP)		●	○		○	-				-	-	-	-		⋆		⋆	
GenVoca		○	○							-	-	-	-			-	-	-
Integrative Composable Generators (ICG)		●	●	⋆	○	●	⋆	○		●	●		●	⋆		⋆	⋆	
Adhoc	-	-	-	-	-	-	-	-		-	-	-	-	-	-	-	-	-
Delegating Compiler Objects (DCO)	○			?	?	●	○			?	?	?	?	?	?	?	?	?
Intentional programming (IP)	●			?	?		·			?	?	?	?	?	?	?	?	?
Jakarta Tool Suite (JTS)		●	●		●	○	·			⋆	⋆	○	●		●			
Functional		○	●	●	●	●	⋆	⋆		⋆	⋆	○	●		●		⋆	

Table 4.3: Overview of the capabilities of each LDT to realize the tasks and challenges of each language concern.

LDT	Strategies	Concern	Task
Rewrite rules	traversals	multiple input	obtention
	rewrite strategies	multiple output	scheduling
	limited form of structure-shy paths	multiple input	obtention
	dynamic rewrite rules	multiple output	identification
		multiple output	integration
Graph rewrite rules	implicit node creation	multiple output	identification
		multiple output	integration
	morphisms	multiple output	identification
		multiple input	obtention
Attribute grammar	attributes	multiple input	identification
	copy- and propagation rules	multiple input	obtention
	attribute forwarding	multiple input	identification
		multiple input	provision
Template-based approaches	structure-shy queries	multiple input	obtention
Compositional Systems	Composition rules	multiple output	integration
	Identification rules	multiple output	identification
	ICG	basic concern	consistency
Ad-hoc	monads	multiple input	obtention
	monads	multiple output	integration
		compositionality	local and global
	traversals	multiple input	obtention
	symbol tables	multiple input	identification

Table 4.4: Overview of the interaction strategies offered by LDTs.

Strategies	Metaoperation	Task
traversals	operations on terms	generic traversals on the parts of any term
limited form of structure-shy paths	operations on terms	generic access and traversals on the parts of any term
dynamic rewrite rules	operations on the set of rules	defining rules and adding rules to the set of rules at runtime
implicit node creation	operations on nodes	generic identification and combination of nodes
morphisms	operation on nodes and connections	generic expressions operating over connections between nodes and nodes themselves
copy- and propagation rules	operation on attributes	generic rules operating on attributes, to compute the values of an attribute using a given set of attributes
attribute forwarding	operation on attributes	intercepting unknown attribute value requests and redirecting it
structure-shy queries	operation on terms	query on the parts and parent of a term
Composition rules	operation on program fragments	composing program fragments
monads	operation on computations	combines computations into a new computation
symbol tables	operation on information	generic storage and retrieval capacities of information

Table 4.5: Overview of the metaoperations performed by interaction strategies.

Chapter 5

Linglets : The basic language concerns

In the previous chapter, we defined a language as a set of cooperating language concerns. Each basic concern, defines in isolation of other concerns, a single language construct by its syntactical definition and its translational semantics. A concern is completely separated from another concern when it adheres to five sets of requirements of our modularization model discussed in Section 4.1. In this chapter, we present the design of a language implementation system which turns these requirements into a solution.

The language implementation system we present in this dissertation is called the *Linglet Transformation System (LTS)*. In LTS, a language implementation is conceived as a set of *interacting* language modules called *linglets*, each capturing a basic language concern. This is called the kernel of LTS. The interactions among linglets using special-purpose concerns are discussed in the next chapter.

A suitable architectural style is chosen to reflect the interactions of the system. The division of a language in basic concerns requires a totally different architecture which is in fact orthogonal to the typical architecture used in contemporary LDTs (see Section 2.2.2).

LTS is implemented in an object-oriented style because this paradigm fits the interactions among linglets. Each linglet represents a program fragment of a larger program and defines several methods which operate on that program fragment. The most important methods of a linglet are its translational semantics, its concern-specific logic, and logic to complete the complex translational semantics of linglets with the necessary interactions and cooperations.

The remainder of this chapter is structured as follows. We first describe the running example which will be used throughout this chapter. Section 5.2 presents the architecture of LTS and introduces its two major parts: linglets and language specifications. We detail the features of LTS that ensure the modularization of the various language concerns into linglets and that enable adaptations of linglets for implementing interactions among them. The language specification of linglets and of language specifications themselves are detailed in Sections 5.3 and 5.4.

Equipped with these specifications we show in Section 5.5 how linglets enforce and adhere to the requirements of our modularization model to ensure their separation. Furthermore, we show that in the language specification linglets can be combined together and adjusted with the translational semantics complements, and the necessary concern-specific logic.

5.1 A Running Example: T2SQL Language

Languages are constructed by composing modular basic language concerns, each defining the syntax and the translational semantics of a single language construct. In order to clarify this, we implement the Tuple Calculus language [Cod72, EN94] in terms of SQL [CAE+76]. We refer to this language implementation as the T2SQL language. T2SQL is used as a running example throughout this chapter. The Tuple Calculus language and SQL are designed for querying databases.

Note that this chapter does not contain the whole implementation of the T2SQL language. We rather focus on the interesting and relevant parts of the language implementation for the purpose of this chapter.

Tuple Calculus Language

The Tuple Calculus Language is a formal declarative language where queries are formulated as predicates. As an example, consider a query to collect the family names of the employees with a wage of 50.000 that work on the project 'Little Boy'. This query is formulated in Tuple Calculus as follows:

```
(1)   { t.family |
(2)     employee(t)  ∧
(3)     t.family = t.lastname  ∧
(4)     t.wage = 50.000  ∧
(5)   ( ∃  w)( workson(w) ∧ w.ssn = t.ssn ∧ w.project = 'Little Boy'
) }
```

Queries in Tuple Calculus are defined as a mathematical sets. A set consists of two parts: a header (line 1) and a condition (lines 2-5) which are separated by a vertical bar. The query collects all tuples `t` of the universe that are valid for a given condition. The tuples in the header are either free or bound variables in the set. They can remain free or can be bound to a relationship. If the tuple variables remain free, the tuple is entirely constructed through the equality equations such as in line 3. If a tuple is bound to a relationship (line 2), the tuple becomes a member of that relationship. Note that equality equations are also valid when the header tuple variables are bound. In that case, the result tuple is extended with the additional values. For instance in our example query, the tuples `t` are

bound to the relationship `employee` and contain an additional attribute called `family`.

The set, defined in the query above, contains all the tuples with a family attribute where `t` is an employee with a wage of 50.000. In addition, for each tuple `t` there exists one tuple `w` of the `workson` relation, with the same social security number (`ssn`) attribute as the employee tuple and with a project attribute equal to `Little Boy`.

The Tuple Calculus grammar in BNF[BBG+60] that we consider is:

```
NamedSet ::= ID "=" Set
Set ::= "{" Attributelist "|" Expression "}"
Attributelist := Attribute
                 | Attribute "," Attributelist
Attribute ::= ID "." ID
Expression ::= Expression Operator Expression
          | NOT Expression
          | Relation
          | BinOperation
          | ForAll
          | Exists
Relation ::= ID "(" ID ")"
Operator ::= AND | OR
BinOperation ::= Attribute BinOperator Attribute
BinOperator ::= EQUALS | GREATERTHAN | ...
Exists ::= "( EXISTS ID ")" "(" Expression  ")"
ForAll ::= "( FORALL ID ")" "(" Expression  ")"
```

The names printed in capital refer to the following lexical categories:

NOT	$\neg$
AND	$\wedge$
OR	$\vee$
EXISTS	$\exists$
FORALL	$\forall$
ID	`[a-zA-Z_@$&#%][a-zA-Z_0-9]*`
EQUALS	$=$
GREATERTHAN	$\geq$

SQL

The target language of the T2SQL language is SQL. SQL is a structured query language operating on a database schema. Below we reformulate the same Tuple Calculus example query in SQL.

```
SELECT DISTINCT t.lastname as family
FROM employee t
WHERE t.wage = 50.000 AND
      EXISTS ( SELECT * FROM workson w
               WHERE w.ssn = t.ssn AND w.project = 'Little Boy')
```

SQL queries consists of a number of clauses. Only the `SELECT`, `FROM` and `WHERE` clauses are considered. SQL `select` queries return a new table called a result set or result table. The columns to be returned are specified in the `SELECT` clause. The example query selects[1] one column of the table `employee` called `lastname`. The name of that column is aliased with the keyword `as` to give it the name `family` in the result table. The table `employee` is aliased with the name `t`. The `WHERE` clause specifies an additional condition stating which rows are selected from the given tables. Only rows with a `wage` column value of `50.000` are selected. In addition, for each `employee` row there must exist at least one row of the `workson` table with the value of the column `ssn` being the same as the value of the column `ssn` of the `employee` row, and the project column being equal to `Little Boy`.

The SQL grammar in EBNF that we consider is:

```
Select ::= "SELECT" ["DISTINCT"] Columnlist
           "FROM" tablelist "WHERE" Expression
Columnlist := Column
              | Column "," Columnlist
Column ::= ID "." ID as ID
Tablelist := ID
             | ID "," TableList
Expression ::= Expression Operator Expression
          | NOT Expression
          | Relation
          | BinOperation
          | Exists
Operator ::= AND | OR
BinOperation ::= Attribute BinOperator Attribute
BinOperator ::= EQUALS | GREATERTHAN | ...
Exists ::= "EXISTS" "(" Select ")"
```

The names printed in capital refer to the following lexical categories:

[1]The technical term for 'selecting' columns is 'projecting'. Projection results in a vertical slice of a table.

```
NOT          NOT
AND          AND
OR           OR
ID           [a-zA-Z_][a-zA-Z_0-9]*
EQUALS       =
GREATERTHAN  >=
```

In the following sections, details about the translational semantics of the T2SQL language are given along with the examples.

5.2 LTS Architecture

The concepts of our new language design technique have been introduced in Section 4.2. Recall that, languages are designed with three kinds of concerns: *basic language concerns* defining the language constructs and *language specifications* defining the interactions between basic concerns by using *special-purpose concerns* which in turn define the mechanisms to implement the interactions.

In LTS, linglets implement the basic language concerns which we defined in Section 4.2.1 and are structured according to Definition 4.11 $\langle \bar{G}, \mathcal{D} \rangle$. A linglet thus consists of two parts: a grammar $\bar{G}$ specifying the syntax of a language construct, and a set of functions $\mathcal{D}$ defining its translational semantics.

In addition, due to the modularization of special-purpose language concerns, concern-specific logic (see Definition 4.15) can be added to linglets. This logic establishes the interactions among linglets. As such, linglets are made responsible for all of their behavior in a language. The special-purpose language concerns form the subject of the next chapter.

The components of our architecture are linglets. The connectors are the interactions among linglets in order to parse the complete source program and in order to produce an equivallent target program. The set of connectors are special-purpose concerns. Architectural specifications correspond to language specifications.

The architectural style of LTS is quite different from the typical LDT architecture (see Section 2.2.2). Pipe and filter architectures use a shared data structure as a common denominator of all components. However, this style does not match the components and their interactions we described above. More precisely, the basic language concerns do not successively process the entire source program text yielding a new target program text at the end of the pipeline.

Therefore we opt for an object-oriented organization [GS93]. An object-oriented architectural organization is a better fit since the components of this style are objects. Objects bundle data along with the possible manipulations (functions and procedures) on that data. As linglets also bundle an expression of a source program together with their manipulations (e.g. producing an equivalent expression in a target language), objects are a natural representation for them.

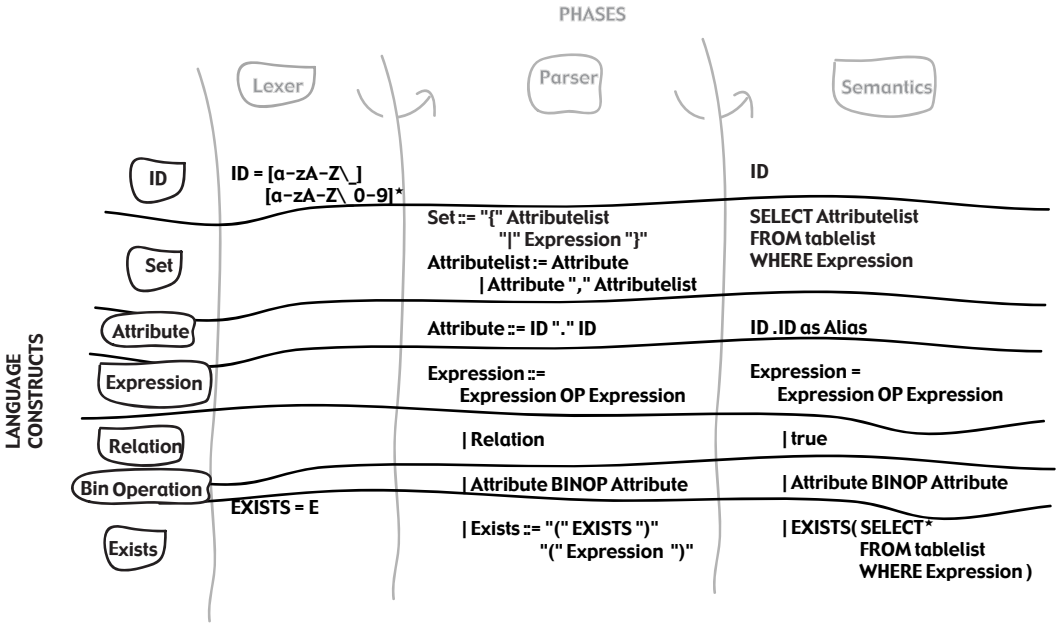

Figure 5.1: Illustration of the decomposition of the T2SQL language using the new architectural style.

We now argue this design choice more precisely by showing that the behavior of linglets nicely maps to objects.

- First, objects bundle data along with their possible manipulations (functions and procedures). Linglets do so too. The data of a linglet is the language algebra $\langle \mathbb{E}, \mathbb{F} \rangle$ given its grammar $\bar{G}$ (see Definition 4.5). The methods on a linglet implement its translational semantics, its concern-specific logic and even its syntactical definition. Recall that the translational complements are not part of the definition of a linglet but they are provided in language specifications (see Section 4.2.3).

- Second, objects interact by sending messages to other objects. These messages invoke one of the methods offered by the receiving object. Linglets behave similarly. Parsing is a coordinated effort among linglets, realized by relying on the parsers of each linglet. In addition, the target language program is constructed by invoking the translational semantics of linglets and their concern-specific logic.

- Third, polymorphism and late binding are powerful mechanisms to structure a compiler. In object-oriented attribute grammars [Hed92, Hed99, Gro92, MuLA99, Paa95], the left hand-side of a production acts as a common superclass for the alternatives in the right hand-side. Consider for example, the production `Expression ::= Assignment`. In an object-oriented attribute grammar, the `Assignment` node class subclasses the `Expression` node class. The superclasses share common behavior. A popular example of such common behavior is the threading of an `env` attribute [MuLA99, Hed99, HM03, Gro92], that models the environment of visible identifiers. Note that although environment threading can be implemented in LTS like the existing LDTs, we treat this interaction mechanism, like any other, as a special-purpose concern. These concerns are discussed in the next chapter.

  Polymorphism and late binding also play an important role in LTS. Linglets are composed in an inheritance hierarchy of language specifications (see Section 5.4.2). In Section 5.9.2, we argue that polymorphism and late binding of prototype-based languages are crucial to maintain the modularization of the translational semantics which produces multiple results. The potential of polymorphism and late binding for specifying interaction strategies is used in the next chapter particularly in Sections 6.2 and 6.3.

  We also show the more classical benefits of polymorphism and late binding, allowing linglets to share common behavior and allowing interactions among linglets to abstract from more concrete linglets. An example illustrating this is given in Section 5.9.2.

In order to clarify our object-oriented architectural style, we illustrate its impact on the implementation of the T2SQL language in Figure 5.1. In conventional compilers, the three phases of a compiler, which are considered in this dissertation, are lexing, parsing and the generation. Because these phases are the primary components in the typical and conventional language implementation architecture, they are shown in grey. These phases serve as a reference point to illustrate the impact of our new architectural style on a language implementation. As a primary component we will use language constructs which crosscut the phases of a conventional compiler. The figure depicts seven language constructs of the T2SQL language: `ID`, `Set`, `Attribute`, `Expression`, `Relation`, `BinOperation`. Each language construct defines part of the vocabulary of the language (a lexer and a grammar) and its translational semantics expressed in SQL. Consider for example the `Set` construct. The `Set` construct contains two parts: a list of attributes and an expression, and is transformed into a `Select` statement in SQL.

Applying an object-oriented architectural style to the design of the T2SQL compiler decomposes it into a set of interacting objects. Each object represents a language concern, contains the necessary information to recognize and parse its

syntax, and defines its semantics to produce its equivalent translational semantics. These objects are defined by linglets. These are composed and glued together in a language specification (cfr. Section 4.2.3). The primary function of a language specification is to compose the grammars of each linglet into the grammar of the language. The secondary function is to complete linglets with behavior to establish the necessary interactions among linglets and to provide a coherent language.

Figure 5.2 depicts the T2SQL language. Each language construct is separately defined by a linglet (depicted by diamonds) in isolation from the rest of the language. A diamond is divided into three layers, depicting the basic parts of a linglet. The top layer contains the name of the linget. The middle layer contains the parser which is depicted by its syntactical description. The bottom layer contains its translational semantics which is depicted by the linglet's equivalent target language expression. The linglets that are brought into a language specification (black square) which is depicted by the grey arrows pointing from the nodes of their composition (the black lines).

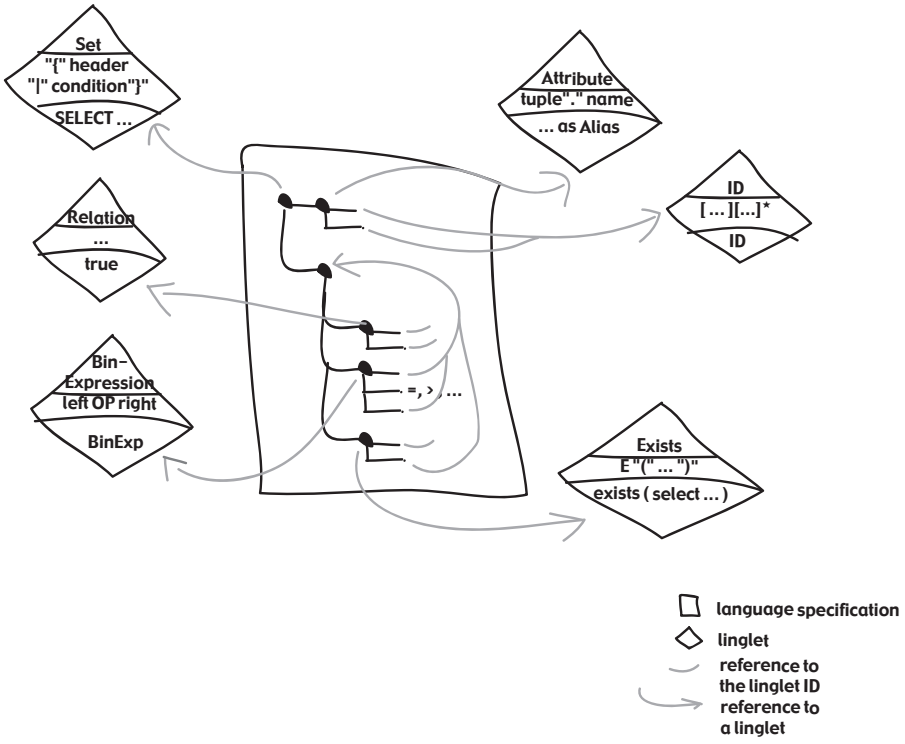

Figure 5.2: T2SQL using an object-oriented architectural style.

5.3 Linglets

Before we commence the explanation of the language defining linglets, we first introduce the major concepts of a linglet by using the example linglet `AttributeEquation`. The `AttributeEquation` linglet defines the language construct to equate two attributes. In the example Tuple Calculus query printed below, the equate construct (line 3) binds an attribute `family` of the free tuple `t` to the attribute `lastname` of the bound tuple `e`.

```
(1)   { t.family |
(2)     ( ∃ e)( employee(e) ∧ e.ssn = '098-00008' ∧
(3)               t.family = e.lastname
(4)   }
```

The `AttributeEquation` linglet is defined as follows:

```
(1)    Linglet AttributeEquation {
(2)       syntax {
(3)          left "=" right
(4)       }
(5)       generate { | left right alias nonlocal local |
(6)         left := ast left
(7)             generate.
(8)         right := ast right
(9)             generate.
10)         ast isFreeTuple
(11)             ifFalse: [ #Expression{ 'left = 'right } ]
(12)             ifTrue: [ local := #Expression{ true  }.
(13)                       alias := right name.
(14)                       nonlocal := #Column{ 'left as 'alias }.
(15)                       local nonlocal add: nonlocal role:#alias.
(16)                       local
(17)             ]
(18)       }
(19)       isFreeTuple { }
(20)    }
```

The syntax of the `AttributeEquation` linglet is defined in its syntactical method called `syntax` (lines 2-4). The syntax consists of three parts: a left attribute denoted by the word `left`, an equal sign denoted by `"="`, and an name attribute denoted by the word `right`. The `left` and `right` are syntactical parameters (cfr. unbound variables in requirement R4 4.5). These parameters are bound to other linglets in the language specification of T2SQL.

The abstract syntax tree (AST) node of the `AttributeEquation` linglet con-

tains two parts or childnodes i.e. `left` and `right`. We refer to the current executing node with the pseudo variable `ast`. The translational semantics of the `AttributeEquation` linglet is defined in its semantical method called `generate`. The translational semantics is compositional so, the linglet retrieves its parts `left` and `right` (lines 6 and 8) by sending the messages `left` and `right`, subsequently retrieves the semantics of its parts (lines 7 and 9) by sending the message `generate` and finally computes a new target program fragment in SQL (lines 10-17).

The semantics of the `AttributeEquation` linglet is context dependent[2]. Depending whether the `left` part of the linglet is a free tuple of the set or not, the `AttributeEquation` linglet produces a different SQL program fragment. The context information cannot be computed by using the parts contained in the `AttributeEquation` linglet. Clearly the linglet requires additional information. The need for that additional information is declared by the abstract method `isFreeTuple` in line 19. This method has to be implemented in the language specification, and has to determine whether or not the `left` attribute is a free tuple.

If the `left` attribute is not a free tuple of the set, then an attribute equation is translated into a SQL expression comparing the two attributes. This expression is produced in line 11 by using the `#`-construct which enables the construction of target programs using the concrete syntax of our target language SQL. The quoted variables are metavariables.

If the `left` attribute of the linglet is a free tuple (line 12), then an attribute equation is translated into two results: an `Expression` and a `Column`. These results must be integrated in various places in the target language program. To illustrate that consider, our tuple query example where the left part of the equation on line 3 is bound to the attribute `t.family` of a free tuple `t`. This tuple query (see below) is translated to a SELECT statement containing the aliased column (line 1) in the `SELECT` clause and the true expression (line 3) in the `WHERE` clause.

```
(1)    SELECT e.lastname as family
(2)    FROM employee e
(3)    WHERE e.ssn = '098-00008' and true
```

The primary or the local result of the `AttributeEquation` linglet is a simple SQL `true` expression (line 12), the other result is a SQL `column` which aliases a column (line 14). This other result is attached as a nonlocal result (line 15) because SQL expressions and columns cannot be combined into a single SQL expression as the columns must end up in the `SELECT` clause instead of the `WHERE`

[2]Note that for clarity reasons we do not fully explain the reasons for the semantics of this linglet. Examples in the next sections will be explained in more detail.

clause.

With polymorphism and late binding of object-orientation many if statements can avoided. From an object-orientation perspective the if statement (`ifTrue:ifFalse:` expression (lines 10-12)) could be replaced by splitting the `AttributeEquation` linglet into three linglets. An abstract `AttributeEquation` linglet with an abstract generate method, and two linglets which delegate to the abstract linglet. The two linglets would each handle a different case of the if statement: a linglet to produce SQL expression comparing the two attributes (line 11) and a linglet to produce the other two results (lines 12-16). From a transformation perspective, this is a bit more difficult to realize because the nodes of the source program tree are created during parsing. At that time, there is not sufficient information to decide which node to take. A couple of solutions have been proposed ranging from dynamic reclassification of objects [Ser99] via rewritable reference attribute grammars [EH04] to plain source tree rewriting (see Sections 3.1 and 3.2). A solution for this problem could be elegantly and easily supported in LTS because of its prototype-based paradigm. In prototype-based languages an object inherits from another object via delegation. The delegation link can be changed or dynamically computed when necessary.

The next two sections introduce the language to specify language specifications and the language to specify linglets. We only explain how local-to-local transformations are implemented. This transformation serves as a simple illustration throughout the discussion of the language specification. Quite a lot of advanced features remain rather abstract as they are used by linglets to cope with more complex translational semantics i.e. compositionality conflicts, requiring external information and producing multiple results. This is explained in more detail in Section 5.5.

5.3.1 Linglet Declaration

Linglets are blueprints of language constructs. Thus, a concrete language construct denoting a fragment of a particular program is represented by an instantiation of the linglet defining the language construct. Linglet instances are called AST nodes. We properly introduce the term in Section 5.3.3.

A declaration of a linglet defines the linglet type[3], and a body which is a list of method declarations:

```
Linglet ::= LINGLET Type  "{" Linglet-body "}"
Type ::= ID
Linglet-body ::= Syntax  Generate  (Method)*
```

[3]By convention, we start the type of a linglet with a capital letter.

There are two mandatory method declarations in each linglet: a `syntax` method and a `generate`. The former is a syntactical method for defining the concrete syntax of a linglet and uses a modularized grammar formalism. Note that only one syntactical method is allowed per object. The latter is a semantical method implementing the translational semantics of the linglet using an imperative object-oriented language.

Besides the `syntax` and the `generate` method, other methods can be defined for various purposes. Methods defined to control the data of a linglet (see Section 5.3.2) come in pairs of getter and setter methods. Methods defined without a body indicate the need for external information, which must be provided by the language designer when using the linglet in a language. Other methods are defined to perform auxiliary computations. Moreover, due to the separation of special-purpose concerns, additional methods are needed to implement concern-specific logic (see Definition 4.15). Also, due to the isolation of linglets, additional methods are needed to complete their translational semantics (see Definition 4.14). For these reasons, linglets and their instantiations must be extendable with behavior. Behavioral extensions are realized through specialization in a prototype-based object-oriented model. Linglets are arranged in a single-inheritance hierarchy via delegation.

Unlike general purpose languages, the data that is manipulated by the semantical methods is not explicitly defined in a linglet. The methods defined in a linglet serve various purposes. Some methods are an intrinsic part of the definition of the linglet such as the method that defines its translational semantics, other methods are added afterwards to respond correctly to requests made by other linglets, yet other methods establish communication and cooperation. These methods have their own logic and their own data which they manage. So a computation performing on a linglet can thus declare only those values that are important for a computation and may ignore the rest. The absence of a central data declaration, reduces the coupling of the various responsibilities of linglets.

5.3.2 Linglet Data

The data of a linglet are its AST children and other data which is referred to or created for the computation of its translational semantics. Data are not explicitly declared but implicitly by the linglet's syntactical and semantical methods. Language constructs or expressions used in these methods that implicitly declare the data will be presented in Section 5.3.3 and Section 5.3.4 respectively. Because linglets are extended with additional behavior in language specifications, therefore new data of a linglet are not fixed at compile time.

The data managed by a linglet, on which the semantical methods operate, are called *parts* or children. Other data like references to other AST nodes are also called children, yielding the datastructure which is used through the translation into a graph. In object-oriented terminology, these parts correspond to instance

variables. Lingets store their parts in open-ended forms that map labels to values. As they are open-ended, there is no upper limit to the number of mappings[4]. Labels are used to store and to retrieve the values (see next section)[5].

The default value of a part is the bottom element i.e. `nil`. As such, linglets can produce incomplete program fragments which are completed during the execution of the transformation by other linglets (see Section 5.6).

The main source of information for the data of linglets is the parsing of program fragments using its syntactical definition. The semantical methods of a linglet mainly retrieve that data. However, there are cases where the data of linglets are also altered. This occurs for example when circular information is computed (more details and examples are given in Section 5.8) or in iterative computations of information to compute the location of nonlocal results (more details and examples are given in Section 5.9).

The current implementation of LTS does not offer support to avoid naming conflicts. These conflicts can be avoided with adequate namespace management. Note that this is not further discussed in this dissertation.

5.3.3 Syntactical Methods

Syntactical methods describe both the concrete syntax and the abstract syntax of linglets.

Concrete Syntax

The concrete syntax of a language construct defined by a linglet is described in a syntactical method. As said before each linglet has only one syntactical method, this method is called `syntax`.

The syntactical definition of a linglet is defined in EBNF according to the requirements we imposed in Section 4.1.8. EBNF is chosen because of its convenient and concise syntax. All the well-known basic operators are available: concatenation (`Sequence`), disjunction (`Alternative`), optional (`Optional`) and repetition (`Kleene Star`) operators. The operators operate on groups of symbols called `Rules`. The rules in LTS are nested. Sequences of rules state that rules follow each other. Alternative rules state that only one of the rules is required. An optional rule over another rule states that the syntax described by the other role is not obligatory. Rules that may occur multiple times are defined by the Kleene Star. The productions describing the syntax of linglets are:[6].

```
Syntax ::= "syntax" "{" Rule "}"
```

[4]Open-ended forms could be implemented with hash tables or lists. Objects are typically implemented with arrays, restricting the number of accessible fields.

[5]By convention, the label of a part consists of lowercase characters.

[6]Symbols enclosed in brackets are optional. Repetitive symbols are denoted by the Kleene Star. Alternatives are separated by a bar and a series of symbols by a space.

```
Rule ::= Sequence | Alternative | Optional
         KleeneStar | CharRange | String | Ref
Sequence ::= ( Rule )*
Alternative ::= Rule ("|" Rule )*
Optional ::= "(" Rule ")!"
KleeneStar ::= "(" Rule ")*"
CharRange ::= INT "-" INT [ Expression ]
String ::= """ [1-38,40-255]* """ [ Expression ]
Ref ::= ID
```

As linglets are modularized, rules do not operate on terminal symbols or nonterminals, but on primitive rules and syntactical parameters respectively. Terminals are not allowed, since each linglet defines its own syntax, syntax that is considered to be a terminal in one linglet is not necessarily a terminal in another linglet. Therefore, a lexer which turns a character stream into a terminal symbol (token) stream is not very useful, and would even erect unnecessary dependencies in the syntactical definitions of linglets as there needs to be a common agreement on the token stream. Instead of terminals, primitive rules are used, such as: `CharRange` and `String`. The former rule accepts every character between the specified range. The latter rule accepts a series of characters. The optional `Expression`, which accompanies these rules, is explained in the next section.

Nonterminals are not allowed because, in order to preserve modularization, only nonterminals defined within a linglet would be allowed to be used in the syntax definition of that linglet. Instead of nonterminals, an indirection is made through syntactical parameters (see the nonterminal `Ref`)[7]. These parameters correspond to the unbound variables U of the grammar $\bar{G} = \langle T, N, S, D, B, V, U \rangle$ (see Definition 4.5) of a language construct. These parameters are bound in the language specification to other linglets (see next section). Hence, the definition of a linglet cannot directly reference other linglets.

Example: The `Set` linglet defines the `Set` language construct of T2SQL (see Section 5.1). The syntax of the `Set` linglet contains two syntactical parameters `variable` and `condition`. These do not directly refer to the linglets `Variable` and `Condition`. In the language specification, a `variable` and a `condition` are bound to the language constructs `Attribute` and `Expression` respectively.

```
linglet Set {
   syntax {
      "{" variable+ "|" condition "}"
   }
}
```

Note that the `+` operator is syntactic sugar for a sequence of a rule followed by a Kleene Star.

[7]By convention, syntactical parameters consists of lowercase characters.

Abstract Syntax

The syntactical description of a linglet does not only define the set of correct language phrases that can be constructed with that linglet, but also implicitly describes the abstract syntax of that linglet. The abstract syntax is a structured representation of the correct language phrases, as it removes all information that does not affect the semantics of a linglet. What remains in the abstract syntax of a linglet is stored as parts.

The nodes in an AST refer to other AST nodes and to primitive data types.

When program text is parsed, the linglet whose syntactical definition matches the program text is instantiated. That instantiated linglet is called an abstract syntax tree (AST) node. Hence, the syntactical description defines a parser, serving as a constructor to create the AST nodes. When a syntactical parameter is encountered, the parser of a linglet invokes the parsers of the linglets bound to that parameter. New AST nodes are constructed and added to the current AST node as parts. So, in a syntactical method, the major parts of the linglet are its syntactical parameters. These parts are stored without a need for semantic actions [App98]. The linglet compiler deduces the applicable data structure required to store the AST nodes. For *multivalued* parameters i.e. parameters that occur multiple times in the language phrases, the AST nodes are stored in an ordered collection. Otherwise, the AST node is stored as a plain reference.

Besides the syntactical parameters, also other parse data need to be stored, namely the values of the primitive syntax rules `CharRange` and `String` (see previous section). Because the system cannot automagically deduce what information to store and how to store it, the linglet developer must manually provide code for this.

Example: In T2SQL, an AST node created by the `Set` linglet (depicted in Figure 5.3) contains a `variable` and a `condition` part, one for each syntactical parameter. The `variable` is a collection of references as more than one `variable` is allowed in the header of a set. The `condition` is a plain reference. The parameters `variable` and `condition` are bound to the linglets `Attribute` and `Expression` respectively. The `Set` linglet contains a parser recognizing its given syntax. When the syntactical parameter `variable` is encountered, the linglet invokes the parser of the `Attribute` linglet and subsequently stores the resulting AST node in the part `variable`. Similarly, when `condition` is encountered, the linglet invokes the parser of the `Expression` and subsequently stores the resulting AST node in the part `condition`.

Merits of a Modularized Syntax

The syntax of language constructs is modularized in LTS by decoupling their syntax definitions with syntactical parameters. The modularization of syntax has several merits. The first merit is the resulting fundamentally different design

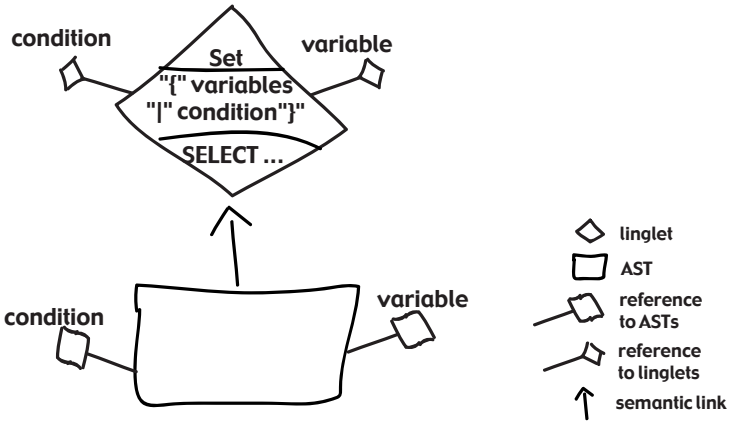

Figure 5.3: An instantiated `Set` linglet of the T2SQL language.

perspective when defining a linglet: it forces developers to conceive the linglet in complete isolation from other language constructs and thus not to depend on syntactical nor semantical properties of other language constructs. This is particularly important for the modularized definition of translational semantics, as the semantics of a language construct is expressed in terms (of the semantics) of other language constructs i.e. its parts (see Section 4.1.3). The syntactical properties of language constructs stem from its syntax definition and are the various parts of the linglets defining these constructs. The semantical properties of language constructs are the various methods supported in the linglets defining these constructs e.g. the kind of return values returned by the method defining the translational semantics, the existence of auxiliary methods to compute additional context information. By not relying on these syntactical and semantical properties, the semantics of language constructs does not depend on the availability of certain parts of other linglets nor on any method supported by other linglets. Consequently, the language in which linglets are used can evolve more easily as linglets do not contain embedded fixed dependencies.

In order to illustrate the gained reusability of linglets let us look at a particular kind of linglets that solve compositionality problems, which are auxiliary language-independent linglets. An example of such a linglet is the `AsNonlocal` linglet. The syntactical definition of this linglet (shown below) consists of one syntactical parameter `body`. The semantics of this linglet, which we cannot show at this point, resolves compositionality conflicts by declaring the semantics of the linglet bound to the body parameter as a nonlocal result i.e. a result which must not directly be used, but must be integrated in another location in the target program.

```
Linglet AsNonlocal {
    syntax { body }
    ...
}
```

A second merit of modularizing the syntax of language constructs is the reusability of syntactical patterns in a language. An example of such a common pattern are sequences. They are used for sequences of statements separated by a dot or a comma, sequences of arguments or parameters separated by a comma, array initializers separated by a comma, etc. In LTS, such a sequence can be fairly easily be defined as a reusable syntactical pattern in the `Sequence` linglet (shown below). This linglet contains two syntactical parameters `element` and `separator` respectively denoting the linglet defining the construct to be sequenced and the separator used between the elements.

```
Linglet Sequence {
 syntax {
   element (separator element)*
 }
 ...
}
```

Also syntactical patterns which are specific for a language can be used. For example, one of the common syntactical patterns in XML formatted languages are tags. Describing the concrete tags in such a language is tedious e.g. the syntax of the HTML tag of the HTML language using EBNF:

```
HTMLTag ::= "<html [ HTMLAttributes ] ">" [ HTMLBody ] "</html>"
HTMLTag ::= "<html" [ HTMLAttributes ] "/>"
...
```

In LTS, the syntax pattern of a tag can be described by the linglet `Tag` (shown below). This linglet serves as a constructor to describe the various concrete tags in an XML formatted language. It consists of four syntactical parameters `lefttagname`, `attribute`, `body` and `righttagname`. In order to construct a specific tag, these syntactical parameters are bound to specific linglets in a language specification.

```
Linglet Tag {
   syntax {
       "<" lefttagname ( (attribute)* ">"  (body)*
       "</" righttagname ">" | "/>" )
   }
}
```

Another kind of example, illustrating the reusability of syntactical patterns, are the auxiliary language independent linglets that solve composition deficit problems. A *composition deficit* arises when two or more linglets need to be composed but have too few syntactical parameters to compose them. A concrete example of a composition deficit and the language independent linglet which resolves it, is shown in our validation in Section 7.4.9.

5.3.4 Semantical Methods

The most important semantical method is the method implementing the translational semantics. In addition, there are accessor methods to manage the parts of linglets, i.e. primarily to retrieve the parts and to ensure consistency when they are altered.

Due to the separation of linglets, additional methods are needed to implement the concern-specific logic and the translational semantics complements. The complements indicate their need for external information (see 5.8) and return and handle multiple results (see 5.9). The latter are discussed in more detail in Section 5.4.2. In order to maintain the isolation of linglets and at the same time ensure their high cohesion, linglets (see Section 5.6.4) and their instances (see Section 5.9.2) must be extendable with such behavior.

Semantical methods of linglets fulfill many tasks and purposes. In this section we discuss how methods can be declared, the primary uses of method declarations, the accessors of parts, the need for external information and the translational semantics of a linglet.

Method Declaration

The translational semantics are declared by an imperative general purpose language. In this dissertation we discuss an implementation of LTS which uses a Smalltalk variant.

Each method of a linglet defines how instances of this linglet will respond to a particular message. As in Smalltalk [Bud86], the particular method being defined is given by the signature of the message. The signature of a method is either a unary selector, a binary selector or a series of keyword selectors. An unary selector is a simple identifier. A binary selector takes two arguments, and is used to model operators as methods. A keyword selector is an identifier followed by a colon. After each keyword selector an identifier denoting the parameter variable must be specified.

```
Method ::= MethodSignature
           "{" [ Temporaries ] Statements "}"
MethodSignature ::= Name  [":" Parameter (Name ":" Parameter)* ].
Temporaries ::= "|" ID* "|"
Statements ::= ...
```

Due to implementation issues we excluded a `return` statement. At the end of a method the result of the last expression is returned. Because linglets implement a prototype-base object-oriented language, two new pseudo variables have been added. They are called `ast` and `previous`, and correspond to the `self` and `super` pseudo variables of Smalltalk[8]. Unlike normal variables they do not need to be declared, they are made available by the system. Both `ast` and `previous` refer to the currently executing linglet instance. When a message is sent to `ast`, the message is looked up in the current linglet instance. On the other hand, when a message is sent to `previous`, the search for a method begins in the delegate of the current linglet instance. Delegation is discussed more in detail in Section 5.3.8.

Accessors

The parts of a linglet are not directly accessible. Although the form is open-ended, it is strongly encapsulated. The values stored in AST nodes are only accessible and changeable via a pair of (inspectors or) getters and (mutators or) setters. Also linglet methods cannot access the values directly.

Setters and getters are semantical methods with the following signature *`part:`* and *`part`* respectively. For example, in order to access the part, say `bar`, the method `bar` is provided, and to alter the part the method `bar:` is provided.

A default getter and setter are provided by the linglet compiler. So a new part, say `foo`, can be added by the default setter `foo:`. As such, strong encapsulation (see Section 5.6 and Section 5.7) is successfully combined with the openness of linglets (see Section 5.8.2).

Example: The `Set` linglet has at least two possible parts `variable` and `condition`. These are implicitly declared by the syntax of the linglet. In the code snapshot below, the `variable` part is accessed and a new part `unique` is created and initialized to `true`.

```
variable := ast variable.
ast unique: true.
```

The strong encapsulation of parts is a prerequisite in order to ensure local consistency and to tackle localized compositionality conflicts. The former requires to specialize their corresponding setters (see Section 5.7), the latter requires to specialize their corresponding getters (see Section 5.6.4).

Example: The code fragment below shows how to override the default getter and setter of the part `unique`. The getter `unique` always returns the value `true`. The setter `unique:` is implemented in terms of the default setter.

```
unique {
```

[8]The Smalltalk pseudo variables `self` and `super` are excluded by the linglet compiler, as they operate in the Smalltalk runtime not in the LTS runtime.

```
    true
}
unique:value {
   value isTrue: [ previous unique:true ].
   true.
}
```

External Information

The external information required by a linglet is explicitly defined by a method. When the information is not available or has no default value, the method is defined with an empty method body (a.k.a. an abstract method). The method is named after the information that is desired.

```
unique { }
```

There are no restrictions on the signature of methods. However, when the signature equals the signature of a getter method of a syntactical parameter, then that part is both considered as a syntactical parameter to be bound to a linglet and as a request for external information. The value of that part will only be accessible when a message is sent via its `previous`.

Translational Semantics

The translational semantics are defined in a single semantic method called `generate`. As is required by the compositionatility requirement R1a, the semantically equivalent target language expression is produced by using the translational semantics of their parts, and by using information external to the linglet.

The produced target program is also represented as an abstract syntax tree to allow future manipulations to be invoked by other linglets (i.e. integration of multiple results, completion of partial results, etc.).

The trees representing the target language program fragments can be constructed manually by composing and instantiating linglets or via a `#`-construct (see Section 5.3.6) by using the concrete syntax of the target language.

More complex translational semantics produce multiple target language fragments. These fragments are either returned as a plain set or as a combined single result. In the latter case, the returned result is called the primary result. The other results are hooked on that primary result as nonlocals (see Section 5.3.7).

5.3.5 Standard Namespace `base`

LTS offers a namespace `base`. That namespace bundles a number of pseudo variables which refer to the linglets that are defined in the target language of the current language specification. The pseudo variables can be accessed with the

dot-operator. These variables are used to construct AST nodes. In the example below, the `base` namespace is used to access the `Select` linglet of the target language SQL.

```
selectnode := base.Select new.
```
[9]

5.3.6 #-Construct

The #-construct is quite similar to the quasiquote construct [Gra94] of LISP, to templates in template languages such as Velocity [Vel03], and to rewrite rules using concrete syntax of ASF+SDF [vdBK02]. At the end of the section, we detail the differences between #-construct and other approaches.

The tree representing the target program fragment can be constructed by hand in LTS.

Example: The semantical equivalent of the `Set` linglet, shown below, is a select query which is defined in its `generate` method. The first two lines of the `generate` method retrieve the translational semantics of the parts `variable` and `condition` of the current `Set` node, by invoking the `generate` method on their value. The two local variables `variable` and `condition` contain the columns of the equivalent `SELECT` query and the condition of the `SELECT` query respectively. The table of the `SELECT` query is called `dual`. The node is created by sending the `new` message to `Table` linglet (line 5). The body of the `table` is an `ID` node (line 3) which contains the string `dual` (line 4). The `Select` node is created in line 7. The `Select` node is subsequently composed with the nodes `variable`, `condition` and `table`. The composition with the former two is performed using the appropriate setter on the `Select` node. The composition with the later is performed by adding the `table` node.

```
     Linglet Set {
        syntax {
           "{" variable* "|" condition "}"
        }
        generate { | variable condition dual table query |
(1)      variable := ast variable generate.
(2)      condition := ast condition generate.
(3)      dual := base.ID new.
(4)      dual body: 'dual'.
(5)      table := base.Table new.
(6)      table body: dual.
(7)      query := base.Select new.
```

[9]The linglets and the `new` method are part of the metalayer of LTS, which is discussed in Section 6.1.

```
(8)       query column: variable.
(9)       query where: condition.
(10)      query from add: table.
(11)      query.
         }
     }
```

Constructing the tree representing the target program fragment by hand is not only tedious and error-prone, but also requires detailed knowledge of the structure of target programs. As domain-specific languages(DSL) often gradually increase the abstraction level, such detailed knowledge is troublesome in an incremental language development process in which DSLs are chained so that a DSL is a source language and a target language both at the same time. In an incremental development process where a language changes at each iteration or increment, the target language is thus also subject to change. As the construction of programs of such languages by manually creating an AST is fragile, LTS features a special language construct `#` to produce the AST by using the concrete syntax of the target language. With the `#`-construct no code has to be manually written to create an AST of a target program. The construct not only facilitates the construction of target programs, it also significantly increases the readability and comprehensibility, since the developer immediately recognize the actual expression being produced.

The developer can easily and quickly respond to changes to the language specification of the target language. If the language is merely extended with optional syntactical clauses then existing program fragments described using the concrete syntax remain valid. If changes are made in the syntax being used, then the target language expression can easily be updated by the developer to reflect the new target language syntax.

The `#`-construct (see below) parses a program fragment and handles escaping variables (a.k.a. metavariables). The `ID` indicates the type of a linglet, so that the parser can disambiguate the fragment by invoking the parser of the proper linglet. The target language expression is formalized by the `Hash-body` production and is enclosed by curly braces. These outer curly braces are part of the syntax of the `#`-construct, the inner curly braces (by nesting the Hash-Body) are considered part of the target language expression. The metavariables in the expression are defined by the `EscapingVariable` production. By quoting[10] a variable, the resulting AST refers to the value of the metavariable rather than their names being interpreted as plain target language expressions.

[10]Escaping to the computational environment

```
Hash ::= "#" ID Hash-Body
Hash-Body ::= "{" ( EscapingVariable | Hash-Body
                  | [0-124,126-255] )* "}"
EscapingVariable ::= "'" ID
```

Note that the metavariables may either refer to a single AST node or to a set of AST nodes.

Example: The `#`-construct allows us to construct the `Select` query using the syntax of the SQL language more easily. We use two metavariables `variable` and `condition`. The produced `Select` AST refers to the content of those variables.

```
Linglet Set {
   syntax {
      "{" variable* "|" condition "}"
   }
   generate { | variable condition |
    variable := ast variable generate.
    condition := ast condition generate.
    #Select{ SELECT DISTINCT 'variable FROM dual WHERE 'condition }
   }
}
```

The `#`-construct has some limitations. A target language expression can only be parameterized with metavariables. More complex fragments have to be created manually.

The `#`-construct is similar to quasiquote construct [Gra94] of LISP with the difference that the syntax of the quasiquoted program can be an arbitrary language. There is no special operator such as the `,@` operator to unquote a list of AST nodes. Sets of AST nodes or references to nodes are both unquoted with the same `'` operator. The `#`-construct is also similar to templates in template-based programming languages (see Section 3.4). Template languages such as Velocity [Vel03] often offer language constructs to embed more complex expressions than plain metavariables and in addition offer language constructs for control flow `if` and looping constructs. In ASF+SDF [vdBK02] the `#` can be transparently used i.e. the left hand side and right hand side of rewrite rules may be written in the syntax of the target language.

5.3.7 Standard Part `nonlocals`

Linglets can produce multiple results which must be integrated at different places in the target program. The linglet that uses produced program fragments is called the using linglet. A using linglet distinguishes between local and nonlocal results, as it does not accept arbitrary AST nodes because the composition of AST nodes

in the produced program fragment must yield a grammatically and semantically valid program fragment. Nodes that yield a semantically and syntactically valid program fragment in the using linglet are called the locals, others are called the nonlocals. By classifying the AST nodes into locals and nonlocals, nonlocals can be passed to a linglet without a linglet having logic to handle them. The modularization of linglets is guaranteed as linglets can now produce results which cannot be immediately handled by using linglets.

Each AST node has a special part called *nonlocals* to store an arbitrary number of nonlocal results. Nonlocals can only be accessed but not altered. In other words, there is no setter for the *nonlocals* part.

Beside the ability to separate what can be handled and what not, hooking the nonlocal to the locals is also beneficial because it captures the proper context to further process nonlocals (see Section 5.9.2).

The `nonlocals` part is accessed via the primitive `nonlocals` method which returns a set which understands the same methods of those of an ordered collection. Nonlocals are added to the set by the methods `nonlocals:role:` and `nonlocals:` (see below). The `role` argument is a name of a role which is used to identify nonlocals on a more abstract level.

```
nonlocals: anAST role: aRole
nonlocals: anAST
```

5.3.8 Specialization

Since linglets are implemented with a prototype-based object-oriented model, specialization is implemented with delegation.

In LTS, delegation is used to extend the data, the behavior, the syntactical parameters and to override the syntactical methods.

The behavior defined in a delegatee[11] is applicable at its delegator[12]. A message that is sent to a delegator is first resolved at the delegator. If a method matches the signature of the message, the method is executed. Otherwise, the message is forwarded to its delegatee. If a method matches the signature of the message, the method is executed. This process continues until there are no delegatees left or an appropriate method has been found. If no method has been found in the delegate chain, then the message `unkownrequest:on:args:` is executed. The default behavior of this message throws a compilation error.

Syntactical methods always override a previous definitions. In other words, new syntax defined in a specializing linglet cancels previous syntax definitions. However, syntactical parameters which are implicitly declared remain a part of the definition of a linglet, and cannot be cancelled out. Thus specialization of syntax

[11] A delegatee is the object to whom requests are delegated.

[12] A delegator is the object that delegates requests to its delegatee.

can only extend syntactical parameters and change concrete syntax. Hence, new syntax must contain the already existing syntactical parameters.

Recall that data is implicitly declared and modeled as an open-ended form in linglets. Data cannot be shadowed or cancelled. More details about the relationship between data and delegation are given in following sections.

For those who are not familiar with prototype-based object-orientation, we need to stress the fact that everything is an object. Linglets are specialized with other linglets, linglets are specialized into instances and instances are specialized with other instances.

Specialization of Linglets

Linglets are combined in the language specification in order to define a language. At that point, linglets can be specialized with other linglets. This is discussed in Section 5.4.

Specialization of Instances

Linglet instances are specialized by declaring a series of methods following an expression yielding a linglet instance.

```
ASTSpecialization ::= Expression (Method)*
```

Example: Reconsider the `Set` linglet. Its semantical equivalent is a `select` query returning distinct elements. The linglet instance or AST node representing the query is stored in the temporary variable `query`. The `distinct:value` method of this `Select` node is overridden to ensure that its `DISTINCT` clause is not removed. As such, the produced query always returns a set of rows.

```
Linglet Set {
   syntax {
      "{" variable* "|" condition "}"
   }
   generate { | variable condition query |
    variable := ast variable generate.
    condition := ast condition generate.
    query := #Select{ SELECT DISTINCT 'variable
                      FROM dual
                      WHERE 'condition }.
    query distinct: value {
     (value == #on) ifTrue: [
         previous distinct: value
      ]
    }.
```

```
    query
   }
}
```

When methods can be declared on a linglet instance, the set of acquaintances available within a newly declared method are the parts of that linglet instance. Furthermore, the declared method has also access to the variables of the current run-time environment of the method which declares the declared method. Consider the following example:

```
(1)   | table attribute |
(1)   ...
(2)   table := #Table{ Employee e }.
(3)   attribute := #Attribute { e.name }.
(4)   table alias: newAlias {
(5)          previous alias: newAlias.
(6)          attribute tuple: ast alias
(7)   }
```

In lines 2 and 3 two target program fragments are created: a `Table` AST node which is stored in the local variable `table`, and an `Attribute` AST node which is stored in the local variable `attribute`. In line 4 the setter `alias:` of the `Table` AST node is specialized so that upon changing the `alias` of the table, the `tuple` of the `Attribute` AST that refers to the `Table` `alias` is updated as well. This is implemented in two steps. First the alias of the `Table` AST node is updated in line 5. Depending on the logic of the `previous` method this alias is either updated or not. To keep this into account, the `alias` of current `Table` AST node is retrieved instead of using the `newAlias` variable, and is used to change the `tuple` of the `Attribute` AST node. Clearly, when specializing the `alias:` method of the linglet instance `table`, within this specialized method the `ast` and `previous` pseudo variables refer to the linglet instance `table` being specialized and the variable `attribute` defined in the current run-time environment can be accessed.

Information lookup: Access to variables of the current run-time environment makes information available such as local variables containing: the translational semantics of the parts of the defining AST node, new target program fragments and other derived information. As such, information at the definition site of a method can be used across the entire target program, without having to compute complex queries to retrieve information from that definition site. Such queries would create hazardous dependencies between linglets in an incremental language development process. An example illustrating this is given in Section 5.9.2 where a method declared on a nonlocal node refers to a local variable from its definition site in order to decide where the nonlocal in the target AST should be integrated.

Delegation: Plain application of delegation at linglet instances causes update consistency problems [MB97]. Firstly, problems arise due to the loss of object identity. In order to specialize the behavior of an AST node, one often relies on the previous definition of that behavior. In these cases, new AST nodes are created and thus also a new object identity. This means that AST nodes that reference the current object identity are not affected. Therefore, interactions still refer to the AST node's old behavior, which may lead to inconsistencies. LTS prevents such situations by referencing AST nodes via proxies. Hence, identity across multiple specializations remains the same.

Secondly, problems arise due to data retrieval intricacy. AST nodes contain behavior and data i.e. references to other nodes. By chaining the AST nodes with the delegation relationship, each AST node along the chain can contain its own data. Consequently, data can be shadowed and made invisible to previous behavioral extensions (AST nodes higher up in the delegate chain). In LTS, this distribution of data is prevented by centralizing data in the proxy and by restricting delegation to behavior only. Moreover, because object identity is preserved, the non-distribution of data ensures that updates are always made consistent with the current set of behavior extensions. In the language Gilgul, a similar implementation technique is used which is called Identity Through Indirection [KC86].

5.4 Language Specification

The individual linglets of a langauge each define a single language construct. In order to construct a language, linglets are composed together in a Language Specification (LS). As linglets combine syntactical descriptions with translational semantics, the composition determines both the grammar and the semantics of the overall language.

A language specification is divided into two sections: a header and linglet introductions. The header defines the name and specifies the name of the target language. The name of the target language is indicated by the keyword `base`. The first introduced linglet is the root linglet. This linglet acts as a starting point for the grammar and for the transformation process.

```
LanguageSpec ::= ID "base" ID LingletIntroduction*
```

An introduction of a linglet consists of the type of a linglet, the compositions with other linglets through their syntactical parameters and additional behavior to further specialize them.

```
LingletIntroduction ::= String
LingletIntroduction ::= ID SyntacticalParameter* Method* "."
```

Example: The most rudimentary version of the language specification for the T2SQL is shown below. The language is named `T2SQL` and its translational semantics is expressed with the `SQL` language. Other than that, this version of the language recognizes the empty query i.e. the single phrase `"{}"`.

```
T2SQL
base SQL

"{}"
```

The next two sections discuss how the grammar of the language is specified and how the overall semantics of the language is formulated.

5.4.1 Grammar

In order to define the grammar of a language, linglets are introduced by using their type. Recall that types are the names of linglets specified in their definition (see Section 5.3.1).

The grammar of a language is defined by binding the syntactical parameters of a linglet using the colon operator to other linglets (see `SyntacticalParameter` production).

```
SyntacticalParameter ::= ID ":" LingletIntroduction*
```

Example: Consider a snapshot of the LS specification introducing and binding the `Set` linglet. Its syntactical parameter `variable` is bound to the linglet `Attribute`, and `condition` is bound to `Expression`.

```
T2SQL
base SQL

Set
   variable: Attribute.
   condition: Expression.
```

Note that according to the grammar of language specifications, linglet introductions can be nested. The following excerpt of the language specification of T2SQL shows how the introductions of the `Set`, `Attribute`, `ID` and `Expression` linglets can be nested.

```
T2SQL
base SQL

Set
```

```
variable: Attribute
       name: ID.
       tuple: ID..
condition: Expression..
```

The above specification should be read as follows. The `Set` linglet is composed with the `Attribute` and the `Expression` linglet by binding its `variable` and `condition` parameter. In turn, the `Attribute` linglet is composed with the `ID` linglet by binding its `name` and `tuple` parameter. As every linglet introduction ends with a dot, nested linglet introductions end with several dots: in order to terminate the `Attribute` and the `ID` linglet two dots are required, and in order to terminate the `Expression` and the `Set` linglet again two dots are required [13].

Observe that several linglets can be bound to a single syntactical parameter. These are usually treated as alternatives. This is explained in more detail in the next subsection.

For simple linglets that consist of a fixed string as syntax only and that do not have any semantic value, a shorthand is provided. There is no need to define the linglet, the fixed string can be used directly as a literal in a language specification.

An introduced linglet is called a bound linglet, otherwise it is called unbound. An introduction can either refer (by using the type of a linglet) to a previously introduced linglet or to a newly introduced linglet. When all the syntactical parameters of a linglet are bound, then an introduction reintroduces this linglet as an newly introduced linglet. In fact, this allows us introduce linglets several times in a language specification. When not all syntactical parameters are bound, then the introduction reuses an already (partially) bound linglet and binds its unbound syntactical parameters and rebinds its syntactical parameters. For a language specification to be valid, all syntactical parameters of every introduced linglet need to be bound, except if that linglet is not used in a binding of a syntactical parameter.

Example: Introducing linglets several times in a language specification is shown in Section 5.4.1, where the `BinExpression` linglet is used to define a binary boolean expression and a binary comparator expression. Multiple introductions are also useful to be able to use auxiliary language-independent linglets and use linglets that define syntactical patterns (see Section 5.3.3) as they might occur

[13] The watchful reader may have noticed that a additional dot is missing at the end of the previous language specification because every linglet introduction ends with a dot. As the `Set` linglet as well as the `Expression` linglet is introduced, two dots are necessary: one dot to terminate each linglet introduction. For presentation purposes we removed the additional dots. When linglet introductions are nested these dots cannot be removed as that would sacrifice readability.

several times in a language. An example of the latter are sequences which are used to define SQL: a `SELECT` clause contains a sequence of columns, and the `FROM` clause a sequence of tables, both separated with a comma. This can be easily defined by reusing the pattern of a sequence which is defined in the `Sequence` linglet (see Section 5.3.3). An excerpt of the language specification for SQL where the pattern of a sequence is reused by introducing the `Sequence` linglet two times:

```
SQL
base SQL

Select
   select: Sequence
              separator: ",".
              body: Column..
   from: Sequence:
            separator: ",".
            body: Table..
```

This specification can be further optimized in terms of reuse by reusing a partially bound linglet `Sequence` where its separator is already defined. Such a language specification is shown below. At the bottom of this language specification, the partially bound `Sequence` linglet is introduced where only the `separator` parameter of the `Sequence` linglet is bound. This linglet introduction is then reused in the introduction of the `Select` linglet.

```
SQL
base SQL


Select
   select: Sequence
              body: Column..
   from: Sequence:
            body: Table..

Sequence
   separator: ",".
```

Binding Semantics

A single syntactical parameter can be bound to several linglets. Note that if one parameter is bound several times (i.e. multiple bindings) this does not lead to syntax repetition (i.e. the syntax of the bound linglets does not occur multiple

times). This has to be specified in the syntax definition by a Kleene Star (see Section 5.3.3).

The semantics of bindings of syntactical parameters are determined by the linglet declaring them. By default, bindings are treated as alternatives which corresponds to the notion of alternative productions in (E)BNF[14].

The binding semantics of its syntactical parameters specified by a linglet can be circumvented in the language specification through an indirection. We bind the parameter to a single linglet which in turn binds the actual linglets. The extra linglet in between then determines the binding semantics.

Example: In order to illustrate the need to override the default binding semantics in the language specification, consider the alternative `Set'` linglet given below. Its syntactical definition is slightly different from the `Set` linglet (aforementioned). The `Set'` linglet defines a syntactical parameter `header` instead of a parameter `variable`. Moreover, the `Set'` only allows a single syntactical clause defined by the linglet bound to the syntactical parameter `header`. In other words, the syntax of the linglet bound to the syntactical parameter `header` can only occur once.

```
Linglet Set' {
   syntax {
      "{" header "|" condition "}"
   }
   generate { | header condition |
      header := ast header generate.
      condition := ast condition generate.
      #Select{ SELECT DISTINCT 'header FROM dual WHERE 'condition }
   }
}
```

We cannot plainly use this alternative `Set'` linglet to define the SQL language, as the header of set should a list of attributes instead of a single attribute. So in order to use this alternative `Set'` linglet to define the SQL language, we need to override the default binding semantics of the header parameter in the language specification. We do this by a simple indirection using the linglet `Multiple`. The language specification using the `Set'` linglet (see below) binds the `header` to the linglet `Multiple` and subsequently binds the `body` of that linglet to the `Attribute`.

```
T2SQL
base SQL
```

[14] A special syntactical operator exists to treat bindings as a mandatory sequence instead of alternatives.

```
Set'
   header: Multiple body: Attribute..
   condition: Expression.
```

with:

```
Linglet Multiple {
   syntax {
      body*
   }
   generate {
      ast body generate
   }
}
```

Note that we included the translational semantics of the `Set'` to show that nothing substantial needs to be changed to its translational semantics compared to the `Set` linglet. The only thing that has been changed is the name of the part `variable` into `header`.

Aliasing Linglets

Bound linglets can be aliased with new types. Aliases resolve ambiguous language specifications in case a linglet introduction specializes another linglet which has been introduced multiple times. By aliasing the various multiple introductions, a linglet can specialize the proper introduction unambiguously.

```
Linglet ::= ID "=" Linglet
```

Example: Consider the following snapshot of the LS specification of SQL defining a binary boolean expression and a binary comparator expression. Both expressions are defined by using the `BinExpression` linglet. A binary `BooleanExpression` is an alias for a `BinExpression` linglet whose `left` parameter is composed with the `ComparatorExpression` linglet and whose right parameter is composed with the `BooleanExpression` or the `ComparatorExpression` linglet. The `BooleanExpression` supports two operators `And` and `Or`. Similarly, a binary `ComparatorExpression` is an alias for a `BinExpression` linglet whose `left` and `right` parameters are composed with the `Attribute` linglet and whose `operator` is either the strictly smaller or strictly larger linglet.

```
SQL
base SQL

BooleanExpression=BinExpression
```

```
      left: ComparatorExpression.
      operator: "And" "Or"
      right: ComparatorExpression. BooleanExpression.

ComparatorExpression=BinExpression
      left: Attribute.
      operator: "<" ">"
      right: Attribute.
```

5.4.2 Overall Language Semantics

Composing linglets via their syntactical parameters shapes the overall language semantics, as these define the parts that will be used during the transformation process. The translational semantics of bound linglets will be used to construct the translational semantics of other linglets.

When linglets are introduced in a language they need to be adapted in order to behave consistently, coherently and cooperatively, and their translational semantics needs to be completed to effect their semantics. Therefore, linglets are specialized with additional behavior upon introduction in a language specification by a series of method declarations or by using the pseudo syntactical parameter `extends`. The `extends` is a pseudo parameter, as it is not a part but is the delegatee of a linglet.

Specializing Linglets with Additional Methods

Linglets are specialized by specifying a number of method declarations along with their introduction in a language specification. The methods are necessary to adapt the linglets such that they yield a coherent language specification: to add the concern-specific behavior and to complete the linglets in order to establish the necessary interactions among linglets.

Methods are declared by the colon operator, binding a signature of a method to a body. The body is enclosed by curly braces. Method signatures in the language specification have the same signatures as methods in the linglets (see Section 5.3.4).

```
Method ::= MethodSignature ":"
           "{" [ Tempories ] Statements "}"
MethodSignature ::= Name  [":" Parameter (Name ":" Parameter)* ].
Name ::= ID.
Parameter ::= ID.
Temporaries ::= "|" ID* "|"
Statements ::= ...
```

Example: Minor syntactical differences are present in SQL among different vendors of databases. So upon translation of tuple queries, these syntactical differences need to be taken into account. The syntactical difference we are tackling in this example are wild cards. Some venders have chosen a * while other vendors a % to denote a wild card. Converting from one wild card to another should ideally be specified in the language specification, and should not be hardcoded in some linglet. In the language specification below, an excerpt of the T2SQL language is shown where the * characters are converted to %. The conversion is performed in the specialized getter `right:` on the `ComparatorExpression` linglet.

```
ComparatorExpression=BinExpression
   left: Attribute.
   operator: "<" ">"
   right: String.
   right : {
      res := previous right generate.
      res chars replace: '*' to: '%'.
      res.
   }.
```

Specializing Linglets with the Pseudo Syntactical Parameter `extends`

Another approach to specialize linglets is by explicitly establishing a delegation relationship between them. This is done by binding the pseudo syntactical parameter `extends` to another linglet.

If the delegatee linglet is not specified, linglets delegate to the `Linglet` linglet. `Linglet` is a pseudo variable and is defined by a language specification. It serves as the root of delegation chain i.e. a common delegatee among all the linglets in a language. The `Linglet` can be specialized and aliased like any other linglet. As such, common behavior can be enforced upon all the linglets, and upon multiple subsets of linglets.

Example: Recall that in Tuple Calculus queries are expressed as sets. Consider an extension to the Tuple Calculus to define bags. The set and bag constructs are defined by the `Set` and `Bag` linglet respectively. Both delegate to the `Query` linglet such as they can share common behavior, syntax and bindings. It also allows developers to address sets and bags as a single linglet when defining interactions among the linglets. Another example is given in Section 5.9.2.

```
Set
   extends: Query.
Bag:
   extends: Query.
```

```
Query:
   variable: Attribute.
   condition: Expression.
```

Standard Method `nonlocal`*`role`*`: nonlocal`

In order to respect the isolation of linglets, additional responsibility is added in the language specification, outside the boundaries of the linglets that produce nonlocals, to specify how the produced nonlocals should be integrated. In order to do that nonlocals should be a part of the public interface. For this a method `nonlocal`*`role`*`: nonlocal` is called whenever a nonlocal is produced, where the *`role`* is the `role` of nonlocal (see Section 5.3.7). By overriding this method in the language specification the integration logic can be specified. For a given linget `L` producing a nonlocal result with role `r`, the nonlocal can be integrated by overriding the method `nonlocalr: nonlocal` on the producing linglet `L`:

```
L
   nonlocalr: nonlocal {
    ...
   }
```

In Section 5.9.1 a concrete example is given of a linglet producing a nonlocal, and in Section 5.9.2 we show how this nonlocal can be integrated. The remainder of this discussion explains why a new standard method has been introduced, and the complexities regarding this design choice.

In order to integrate nonlocals, the proper context (i.e. the location in the produced target tree where the nonlocals have been produced) is needed as the integration of nonlocals may depend on that context. As we discussed earlier in Section 5.3.7, this context is retained because nonlocals are hooked to locals, deep into the produced target fragment. However, they also must be part of the public interface so that they can be easily identified such that they can be integrated (see Section 5.9.2).

It does not suffice to simply override the `generate` method at the base level because this breaks the separation of linglets as it requires internal details to identify the produced nonlocals nodes deep within the produced target fragment. Furthermore, accessing nonlocals at that time can already be too late. Once nonlocals are hooked into a local result, deep in an AST, the integration process of nonlocals may already start. However, as the integration of nonlocals is only specified after all results are produced, the integration process would not be correct. Therefore, new methods of the form `nonlocal`*`role`*`: nonlocal` are called by LTS exactly at the time nonlocals are hooked deep into the target AST. As such, nonlocals become a part of the public interface and can retain their context information by still hooking them deep into the local results.

5.4.3 LTS at Work

In this section, we give an overview of a language implementation in LTS.

In order to implement a language in terms of another, both the source and the target language must be implemented in LTS. It is obvious that the source language must be implemented in LTS, but it may be less obvious why this is also the case for the target language. The first reason is that the target program is represented using a dense and highly structured representation. By implementing the target language in LTS, this structure becomes available. The second reason is that the translational semantics of a linglet does not only use the produced target program fragments obtained from their parts, but must also be able to perform computations on that structure so as to be able to express complex compositions (see Section 5.6.4). Flexible and extensible ASTs of language implementations in LTS can be extended with additional semantics to facilitate these computations and maintain the modularization of linglets (see next sections). The third reason is that domain-specific languages (DSL) [vDKV00] often gradually increase the abstraction level. Hence, they are chained so that a DSL is both a source language and a target language at the same time. It is thus natural to implement both source and target language in the system.

The regression from one language to another, less developed, language is not infinite. The compiler at the bottom of the chain is the identity compiler. In LTS terms, the base language and the source language of such a language implementation are the same.

The root linglet of the source language starts parsing the source program text and produces an AST tree of the source program text. Linglets invoke the parsers of the linglets bound to them in the language specification. As such, they collectively produce a tree out of the individual AST nodes which are produced by their parsers. The AST nodes have a semantic link with the linglets that created them. This link ensures that the behavior defined in linglets can be applied on their AST nodes. This allows us to initiate the transformation process by invoking the translational semantics of the root AST node. Each node recursively invokes the translational semantics of the parts to combine them into a valid target program fragment.

Example: To further clarify this process, we schematically depict a transformation process in Figure 5.4 in its final stage in transforming the source program:

```
{ t.lastname | employee(t) ∧ (∃ w)( ...  )  }
```

to the target program:

```
SELECT t.lastname
FROM employee
WHERE true and exists ( SELECT w.* FROM ... WHERE ...)
```

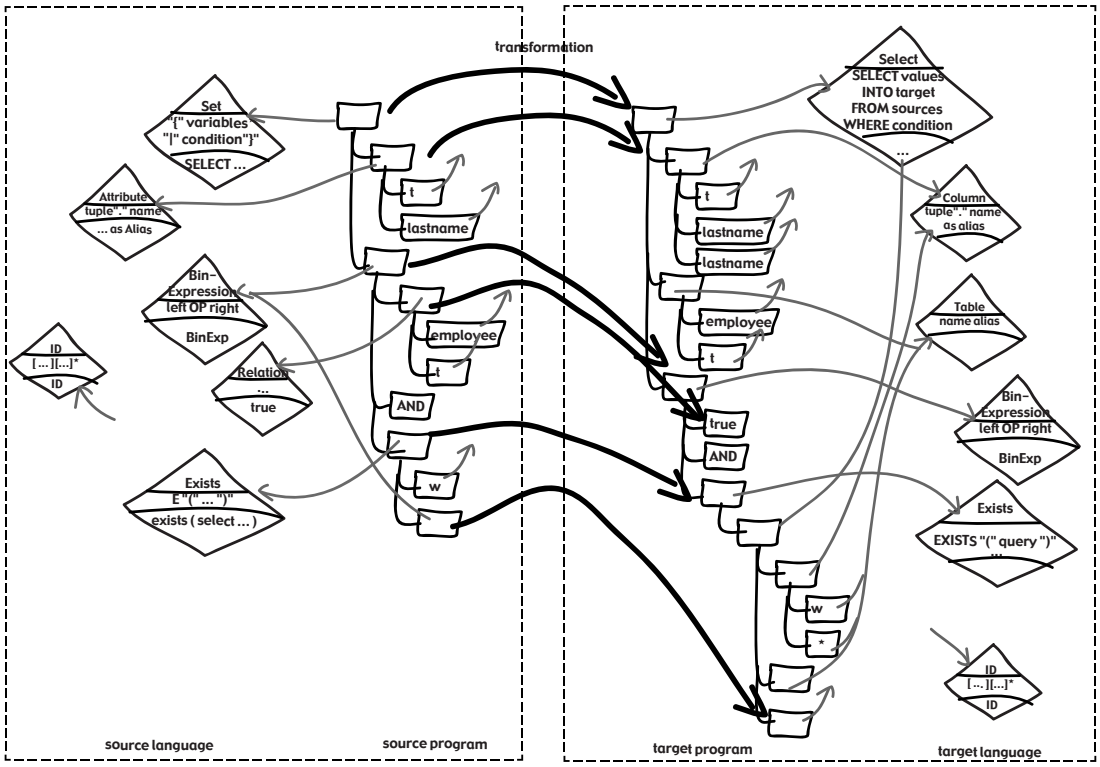

Figure 5.4: Snapshot of the transformation process for a T2SQL program in its final stage.

Note that the ellipsis in the example programs are not part of the syntax of the language but are used in order not to clutter up the figure. Notice also that the target program is not formulated in its most canonical form. The subexpression `true` could be omitted. We deliberately kept that part as the subexpression is the translational semantics of a relation. As such, we are able to depict the translational semantics of each of the T2QL linglets used in the source program.

At the left side of the picture, the source program is depicted. The root node is the `Set` node. It refers to one `Attribute` and a `Binexpression` node. The `Binexpression` node combines the `Relation` node and an `Exists` node with an `AND` node. The grey arrows represent the semantic links between the AST node and their linglets. This source AST is recursively transformed to the target AST, which is depicted on the right side of the figure. The fat black arrows between source and target nodes illustrate the transformations that start at a source node. They point to the nodes that are produced by the transformations. The top node of the target AST is a `Select` node. This node contains three parts: a list of attributes, the tables of the `FROM` clause, and a condition in the `WHERE` clause. The nodes which are not connected to a linglet point to the linglet `ID`.

5.5 LTS Requirements

In Section 4.1, we impose five requirements on the translational semantics of a language construct. Due to the requirements, the semantics of linglets can only be defined but not effected (see Definition 4.8 in Section 4.1.2). Hence, for each of the three kinds of requirements, a special-purpose concern is required offering a mechanism that can be used to effect linglets i.e. implement their interactions. These interactions are defined in the language specification by completing the semantics of linglets.

In the following sections, we revisit each of these requirements and show how LTS enforces and abides them. For each requirement, we subsequently explore how its corresponding special-purpose concerns are used in the language specification.

Note that although the language specification is the sole place where all the linglets of a language and their compositions are known, interactions of linglets are still severely constrained. Linglets can only define their semantics by using the semantics of their parts. A linglet in a plain language specifications can only access the direct or immediate acquaintances through the parts of its AST nodes. Other more distant AST nodes can only be accessed by successively invoking messages to retrieve parts. Such implementations are very fragile in an iterative and incremental development process. In each iteration, any change in the composition of linglets would invalidate such implementations. We therefore introduce interaction strategies to extend the mechanism offered by the language specification in the next chapter.

5.6 R0 - Program Representation

As mentioned in Section 4.1.6, the first requirement enforces that a valuation function produces partial program fragments (R0a). In order to complete the fragments, they need to be completable (R0b). Furthermore, requirement R0c requires that consistency of changes must be controlled by consistency enforcers, local to the produced values and the producing linglet.

5.6.1 R0a - Partial Program Fragments in Linglets

Due to the different abstraction levels of source and target language, we cannot expect that the translational semantics of a linglet expressed in some target language results in a complete and correct target language fragment. The translational semantics of the parts of a phrase may not be sufficient for computing the target language fragment. There are two options for dealing with this problem: one can request external information or one can produce partial values. In order to maintain the separation of linglets, we impose requirement 4.2 stating that only

external information which is essential (see Section 4.6) is allowed to be requested. Therefore, in cases where external information is not essential and is only needed to complete the target language expression, the translational semantics should rather omit these parts and thus produce partial values.

LTS explicitly supports the $\perp$ value to construct partial values. A partial value is an AST node in which certain parts are empty, or technically speaking contains `nil`. Partial AST nodes can be created in two ways: by hand (skipping the initialization of the part involved) or by the `#`-constructor (which constructs an AST representation of target programs using concrete syntax).

Example: Consider again the `Set` linglet whose semantically equivalent SQL expression is a `SELECT` statement. The `Set` linglet is a simple example illustrating the need for partial values. The translational semantics has two target program expressions (`variable` and `condition`) at its disposal to construct a new AST node with three parts (`value`, `source` and `condition`). Clearly, the `Set` AST node has not enough parts to construct its `Select` AST node. Up till now, we circumvented this problem in previous definitions of the `Set` linglet (e.g. Section 5.3.4). In those early definitions, the `source` of the `Select` AST node contained a dummy table called `dual`. Note that it is a correct and valid SQL expression as the `dual` is dummy table within is actually supported in some commercial database management systems[15]. As explained in Section A.1, this solution is ad-hoc, not applicable in all situations, and lacks an explicit intention. Instead of using the `dual` table, we redefine the `Set` linglet using the `nil` value as follows:

```
Linglet Set {
   syntax {
      "{" variable* "|" condition "}"
   }
   generate { | variable condition |
    variable := ast variable generate.
    condition := ast condition generate.
    #Select{ SELECT 'variable FROM 'nil WHERE 'condition }
   }
}
```

In `Set` linglet, `nil` is a metavariable and not a table. By quoting the reserved variable `nil`, that part of the AST is left out[16]. The equivalent definition of the `Set` linglet without the `#`-construct is given below.

```
Linglet Set {
```

[15]The `dual` is a table which is created by Oracle along with the data dictionary. It consists of exactly one column whose name is `dummy` and one record. The value of that record is `'X'`.

[16]If the word `nil` is not escaped it is treated as a plain target language expression.

```
   syntax {
      "{" variable* "|" condition "}"
   }
   generate { | variable condition query |
(1)    variable := ast variable generate.
(2)    condition := ast condition generate.
(3)    query := base.Select new.
(4)    query column: variable.
(5)    query where: condition.
(6)    query.
   }
}
```

5.6.2 R0b - Completable Program Fragments in Linglets

Completable values are an important prerequisite for the definition of a linglet and of special-purpose concerns. AST nodes that represent program fragments are completable because each part of an AST is stored in a slot which is always managed by a setter and an getter method. Only by invoking these methods, the composition of the nodes can be changed.

Example: The `DefineRelation` linglet (shown below) defines the construct for naming a set (see `NamedSet` production in Section 5.1).

```
     Linglet DefineRelation {
        syntax {
           name "=" definition
        }
        generate { | name definition |
         name := ast name generate.
         definition := ast definition generate.
(1)      (definition linglet type = 'Select')
(2)      ifTrue: [ | firstchar |
(3)        definition target: name.
(4)        firstchar := (name asString left: 1) asString toUpper.
(5)        definition values do: [ :el | | oldname |
(6)           oldname := el columnname.
(7)           el columnname: #ID{firstchar'oldname}.
(8)        ].
(9)      ]
(10)     ifFalse: [ CompilerException raiseSignal:
(11)                 'definition is not a Select statement'
(12)                 within: ast ].
(12)     definition.
```

```
        }
    }
```

The translational semantics of a `DefineRelation` linglet is a `SELECT` statement with an `INTO` clause, and subject to the convention that the column names are prefixed with the first character of the name of the relationship.
The following query:
`families = { e.name | employee(e) }`

is thus transformed into:
`SELECT DISTINCT Fname INTO families FROM employee e.`

In order to construct this semantics, a `DefineRelation` AST node uses the semantics of its parts `name` and `definition` which contains an `ID` node and a `Select` node respectively. The implementation of the translational semantics relies on the ability to change the obtained `Select` node. First, we ensure that the semantics of the `definition` part is a `SELECT` statement (line 1), otherwise we throw an exception (line 10). Second, the `DefineRelation` linglet alters the `Select` query, obtained from its part `definition`, to store the result in a table. To this end, we alter the `target` table of the `SELECT` statement (line 4), which is the parameter of the `INTO` clause. In the query example, the target table is the `families` table. Lastly, to enforce the naming convention on the columns, we iterate (line 5) over the `values` and alter the column name (line 7) that will be used in the result table, with a prefix `firstchar`.

5.6.3 R0c - Local Consistency in Linglets

As linglets use the translational semantics of their parts so as to construct their equivalent target program fragment, the interactions among linglets can change any result produced by linglets. In order to prevent invalid changes, linglets get the additional responsibility for ensuring the consistency of their instantiations during successive changes. They do this by overriding their getters and setters, which encapsulate their parts.

Example: In SQL, each column name in a table must be unique, since `SELECT` statements produce new tables containing the selected values, a `Select` AST node must thus ensure the uniqueness of its column names whenever the node is changed.

Consider the `Select` linglet given below. The `value` parameter is bound to the `Column` linglet in the language specification of T2SQL. The consistency of its column names is implemented in the `check` method. This method is called whenever the set of `values` change (intercepted by the `value:` setter method) or when a single `value` is changed in the set (intercepted by `value:put:` setter method). The `check` method renames duplicate `columnnames` to `EXPR`*`n`*, with

`n` an unique number. Note that the `check` method relies on two methods to get and set the `columnname` of duplicated `values`, they are `columnname:` and `columnname`. Hence, the separation of the `Select` linglet is maintained.

```
Linglet Select {
   syntax {
      "SELECT" value "FROM" source "WHERE" condition
   }
   generate { | value source condition |
     value := ast value generate.
     source := ast source generate.
     condition := ast condition generate.
     #Select{ SELECT 'value FROM 'source WHERE 'condition }
   }
   value: index put: value {
     ast check
   }
   value: values {
     ast check
   }
   check { | columnnames index |
       columnnames := Set new.
       index := 1.
       ast value do: [:el |
             (columnnames contains: el columnname asString)
             ifTrue: [ el columnname: #ID{EXPR'index}.
                       index = index + 1 ].
             columnnames add: el columnname asString
      ].
  }
}
```

5.6.4 SP0 - Concern-specific Logic: Cooperation and Coherence in Language Specifications

Accessing and modifying information defined in other linglets to implement interactions by relying on the definition of other linglets would break their modularity. In order to preserve the modularity, interactions should rely on methods which are part of so called *interaction* interfaces. Upon composition and combination of linglets in a language, an implementation for these methods can be added to them. We have called the behavior of these methods the concern-specific logic of a linglet (see also Section 4.3). There are two kinds of concern-specific logic: coherence and cooperation. Note that relying on the interface of the linglets that

define the target language, is a border case, and is not strictly considered as a modularization violation.

SP0 - Concern-specific : cooperation

In order to implement complex translational semantics, linglets rely on external information invoked upon and/or changes applied to the translational semantics of their parts. As linglets should by requirement R1c limit access to parts of nested program fragments, the access and changes are realized by invoking concern-specific logic on the translational semantics of their parts. In order to ensure correct cooperation between linglets, this concern-specific logic is not necessarily part of linglet definitions, therefore it is added to them in language specifications. We first discuss an example of cooperation based on the definition of the linglets of the target language. Afterwards, we discuss the case where language specifications add this logic.

Cooperation by linglets Linglets can use the methods of the linglets which define the target language, to enable cooperation with other linglets.

Example: The most basic form of cooperation was illustrated with the `DefineRelation` linglet, in the subsection completable program fragments on page 188. The translational semantics of that linglet used the default available setter `target:` to change the `target` part of a produced `Select` AST node (line 3 of the definition of the linglet).

Cooperation by the language specification Linglet can also be extended with additional methods in the language specification to support cooperation among them.

Example: The `DefineRelation` linglet and the `Select` linglet use another pair of methods named `columnname` and `columnname:` to prefix the column name and correct duplicated column names respectively. The methods `columnname` and `columnname:` get and set the name of the column, respectively. Those methods are not present in the linglet that define the columns. Columns in SQL are defined by the `Column` linglet (shown below) and the `ComposedColumn`[17] linglet. Given the definition of the `Column` linglet, a `Column` AST node offers three getters and setters, one for each of its three parts: `tablename`, `name` and `alias`.

```
Linglet Column {
   syntax {
      tablename "." name !("as" alias)
   }
   ...
}
```

[17]For conciseness reasons we do not include the definition of a `ComposedColumn` linglet.

Hence, the language specification must define the methods `columnname` and `columnname:` on all the linglets bound to the `value` parameter of the `Select` linglet in order to enable the correct cooperation between the bound linglets. Consider the following excerpt of the language specification of SQL. Three linglets are introduced: `Select`, `Column` and `ComposedColumn`. The latter two linglets are extended with the `columnname` and `columnname:` which inspect and the retrieve the name of the column used for the final result set respectively. The `columnname` of a `Column` used in the result set is the `alias`, while the `columnname` of a `ComposedColumn` is its `name`.

```
Select
values: Column. ComposedColumn.
....
Column
    name: ID.
    tablename: ID.
    alias: ID.
    columnname: { ast alias }.
    columnname: value : { ast alias: value }.
ComposedColumn
    name: ID.
    body: ArithmeticExpression.
    columnname: { ast name }.
    columnname: value : { ast name: value }.
```

SP0 - Concern-specific : coherence

As linglets are defined in isolation, one must ensure that linglets are coherent. Since composing several linglets to form a language does not automagically result in a working language, linglets need to share a common goal to produce a semantically valid target program. We distinguish between implicit and explicit coherence.

Implicit coherence by linglets In most cases there is an implicit agreement among linglets of a language. In other words, linglets are designed to yield a valid target language program.

Example: An example of implicit coherence is the agreement among the `Relation` linglet and the `Attribute` linglet. The former generates the tables which are part of the `FROM` clause of the equivallent `SELECT` statement, e.g. `employee e`. The latter uses those tables in the columns e.g. `e.lastname`. The silent agreement among these two linglets is that the tables are aliased with the tuple name such that the columns can use the tuple name instead of the actual table name.

Explicit coherence by language specifications There are also cases where coherence can be explicitly enforced by changing linglets with additional behavior in a language specification.

Example: Consider again the `Relation` linglet. Its `relation` and `name` parameters are bound to the linglet `ID` in the language specification. Identifiers in Tuple Calculus are syntactically less constrained than identifiers in SQL because in the former, identifiers may contain symbols such as `#`, `@`, `&`, to name a few. So the `ID` linglet of the Tuple Calculus language produces a new `ID` node of the SQL language and prunes the unacceptable symbols. However, whenever the identifier is used as an alias in `SQL` these symbols are allowed, provided that they are enclosed within special markers. Some SQL implementations use double quotes, other use square brackets as markers. These decisions need to be made explicitly coherent across the language implementation during the integration of linglets in the language specification.

In the `Relation` linglet, the `name` part is accessed and used as an alias. As we just explained, when identifiers are used as aliases, the unacceptable symbols do not have to be pruned away. This is resolved in the excerpt language specification shown below. The `name` getter of the `Relation` linglet is overridden to produce a `Text` node containing the original content of the name. When overriding the `name` getter, the original content of the name part, i.e. a T2SQL `ID` AST node, is obtained by invoking the `name` getter on the previous. Instead of using the translational semantics of the T2SQL `ID` AST node, i.e. an SQL `ID` AST node pruned with the unacceptable symbols, the T2SQL `ID` AST node is merely converted to a string. The resulting string is then enclosed by two double quotes and returned as a `Text` node.

```
Relation
   relation: ID.
   name: ID.
   name: { name := previous name asString.
           #Text{ "'name" } }.
```

5.7 R1 - Compositionality

Recall from Section 4.1.3, that requirement R1a enforces that a valuation function operates on language constructs rather than on whole language phrases. It is clear that linglets do so, as they use the translational semantics of their parts to construct their semantically equivalent target fragment. Moreover, compositionality implies that the semantics of a language construct is defined in terms of the semantics of its parts. In other words, linglets assume that the semantics of its parts suffice to yield semantically and syntactically correct program fragments. When that is not possible a compositionality conflict arises, and these conflicts are resolved in the language specification.

Requirement R1c ensures that the translational semantics of a language constructs cannot access the parts of its parts and in turn their parts, etc. LTS does not enforce that requirement but supports it by allowing linglets to request any additional information they need.

5.7.1 R1 - Compositionality in Linglets

To be able to intervene in the composition of linglets, linglets must use getters and setters to access and change their parts. As such, linglets can remain oblivious to compositionality conflicts (R1b). By using getters and setters, interventions in the composition *do not need to change* the composition among the AST nodes. As a result, the effect of the compositions can be kept local and non-destructive.

Example: Consider the language specification shown below and the query in Section 5.1 as sample input to explain a compositionality conflict found in the `Attribute` linglet.

```
Set
   variable: Attribute.
   condition: Expression.
Attribute
   tuple: ID.
   name: ID.
```

The `Attribute` linglet (see below) produces a `Column` node that accesses a column on a table. The produced column is named after the `name` part of the `Attribute` linglet and the table is named after the `tuple` of the `Attribute` linglet. So, the attribute `e.family` in example query is transformed into the column `e.family as family`. This semantics is correct when considering the various linglets of the Tuple Calculus language in isolation. However, when equality equations (cfr. Section 5.1) are used together with attributes, an error occurs. In our example query, it is not correct to produce the name of the attribute `family` in the `SELECT` statement but it should be `t.lastname` which is the name of the attribute bound to it. The error is due to a composition conflict and it occurs in the `Attribute` linglet. If an attribute created via the equality equations is used in the header of a set, then the name of the attribute is *not* the name of its equivalent column of a table. Instead of the name of the attribute, the name of the attribute assigned by the equality equation to the attribute must be used as the column name. Hence, in such cases, there is a semantical compositionality conflict in the `name` part of the `Attribute` linglet.

```
   Linglet Attribute {
      syntax {
          tuple "." name
      }
      generate { | tuple name alias |
(1)       tuple := ast tuple generate.
(2)       name := ast name generate.
(3)       alias := ast alias generate.
(4)       #Column{ 'tuple.'name as 'alias  }
(5)      }
(6)      alias {
(7)          ast name
(8)      }
   }
```

When looking at the definition of the `Attribute` linglet above, notice that the generated `tuple`, `name` and `alias` parts of a `Column` node are obtained by invoking the corresponding getter for each part (lines 1-3). In other words, the parts are not directly accessed. Hence, to resolve the compositionality conflict it suffices to change the composition of the `ID` nodes in the language specification. This is discussed in the following section.

5.7.2 SP1-SP2 - Resolving Compositionality Conflicts in Language Specifications

When linglets are composed, one must make sure that the translational semantics of their parts are syntactically and semantically correct to prevent compositionality conflicts. Compositionality conflicts are resolved by intervening in the composition of linglets (cfr. Section 4.3.1). We distinguished between two kinds of interventions: localized (SP1) and global interventions (SP2). There are two ways to implement localized interventions. The first one is to change the composition of linglets by inserting other linglets. The second one relies on the encapsulation of their parts from their proper methods. Global interventions cannot be tackled by a single basic concern because of its isolation. They can neither be tackled by the language specification because we can only intervene on specific parts but not on all parts. They are tackled with interaction strategies in the next chapter.

SP1 - Insertions between Linglets

The most straightforward way to realize interventions to solve compositionality conflicts is by changing the composition of linglets so that the given composition of linglets do not longer give rise to a conflict. However, one must be careful to

rearrange the composition of linglets as a language specification determines both the grammar and the overall semantics of a language. So, the compositions that affect the grammar are not allowed in LTS. Therefore changes to the composition of linglets are restricted to insertions.
Let `L1` and `L2` be two bound linglets, where:

```
L1 param: L2..
```

Inserting a linglet `L3` results in:

```
L1 param: L3.
L3 body: L2.
```

The inserted linglets must have a single syntactical parameter `body` as its syntax definition, with the appropriate binding semantics to bind the `L2`. As such, the syntax of the inserted linglets do not change the grammar. The translational semantics of these inserted linglets are responsible for resolving the compositionality conflict.

Example: Consider again the `Relation` linglet. In the overview of the LTS system in Figure 5.1 a `Relation` node corresponds to a `True` node and a `Table` declaration node. The `Table` node is the semantical equivalent of a `Relation` node in the SQL. The `True` is produced because a relation is part of a boolean expression in the Tuple Calculus. There are two approaches we can take to implement the `Relation` linglet. In one approach, the linglet produces these two results. In this case, we face a linglet with multiple results. This which will be discussed in Section 5.9. In the other approach, the linglet produces only its equivalent semantics in isolation which is the `Table` node. In that case we face a compositionality problem when a relation is composed within a boolean expression in the language specification. In what follows we discuss how to solve this problem.

The language specification below shows a relation which is composed with a boolean expression. The `BooleanExpression` linglet produces an equivalent `BooleanExpression` node in SQL. The `Relation` linglet is composed with the `BooleanExpression`, so the left or right part of a `BooleanExpression` node can be a `Relation`. However, the `Relation` linglet only produces a `Table` node, but according to the SQL grammar, a table declaration cannot be part of a boolean expression. Hence, there is a compositionality conflict between the `BooleanExpression` and the `Relation` linglet.

```
name T2SQL
base SQL
...
Set variable:
      condition:BooleanExpression.
...
```

```
BooleanExpression=BinExpression
      left: CompExpression. Relation.
      operator: "And" "Or"
      right: BooleanExpression. CompExpression. Relation.
...
```

In order to tackle this composition conflict we insert a new linglet `MakeTrue` between the `BooleanExpression` linglet and the `Relation` linglet by including the following language specification:

```
Relation=MakeTrue
   body: Relation.
Relation
   relation: ID.
   name: ID.
```

The semantics of the `MakeTrue` linglet (shown below) is a `True` SQL node. Due to the insertion of this linglet, the `Relation` linglet can now be used within a boolean expression. More precisely, the semantics of the `Relation` linglet can be composed with the semantics of the `BooleanExpression` because the `MakeTrue` linglet returns a `Table` as nonlocal result. The integration of this nonlocal result is handled by interaction strategies which is discussed in the next chapter.

```
Linglet MakeTrue {
  syntax {
     body
  }
  generate { | res |
     res := #True{}.
     res nonlocals add: ast body generate.
     res.
  }
}
```

SP1 - Encapsulation of Linglets's Parts From their Proper Methods

Another way to intervene in the composition which does not require insertions of new linglets as in the previous option, is by overriding the getters of the parts in the language specification.

Example: In the beginning of this section, we discussed a composition conflict of the `Attribute` linglet when there is an equality equation defining an attribute used in the header of a set. In order to change the composition of the `Attribute` node for resolving this conflict, two new getters for the `name` and the `alias` override the existing composition of the node. The `name` method searches for

the equality equation that assigns the current attribute. If an equation is found, the name of the assigned attribute is used, otherwise no action is taken and the original name is used. Note that by default the alias is the name of the attribute (see line 6-8 of the `Attribute` linglet defined in Section 5.7.1). So, if we change the name of the attribute, the alias also changes. However, the column should remain aliased with the name of the attribute of the set. In order to maintain the desired column name in the result table, we override the getter of the alias to retrieve the original name.

Note that we cannot further discuss the `findAssignedAttribute:name` method at this point, as that method requires to traverse the source program and this is achieved by the interaction strategies.

```
Attribute
   tuple: ID.
   name: ID.
   alias: { previous name }
   name : { | eq |
      eq := findAssignedAttribute: (ast tuple asString)
                                   name: (ast name asString)
      (eq == nil)
         ifTrue: [ previous name ]
         ifFalse: [ eq name ].
   }
```

5.8 R2 - Multiple Inputs

As explained and introduced in Section 4.1.4, requirement R2 enforces that the translational semantics of a linglet can only depend on essential externally provided information next to the syntactical parts defined in the syntax of the language construct defined by the linglet. Hence, the name multiple inputs. Such external information required for the compilation of a term may reside in other concerns or even have to be computed by other concerns.

5.8.1 R2 - Declaring Multiple Inputs in Linglets

In the implementation of a linglet's translational semantics, we do not distinguish between local i.e. its parts, and nonlocal i.e. external input contained in other linglets. Both kinds of input can be retrieved via methods regardless of their location: a part of a linglet or other linglets. Similar to how getter methods provide access to the parts of a linglet, we define additional methods that provide access to external input (R2a). However, because we cannot determine how this additional input is gathered, this method must remain abstract (R2d). Empty

methods are interpreted as abstract methods. They can be specialized in the language specification with the behavior that implements the retrieval of the requested input.

As a linglet can only access its parts, and declare its need for external information, a linglet can only consider one language construct at a time (R2b).

No artificial distinction is made between the different sources of inputs. In essence, the distinction between local and nonlocal input is not important when implementing a linglet and when using a linglet. But the distinction is still clear enough to be able to know what external information should be provided to a linglet. This is not only important for language designers but also for the implementation of special-purpose concerns discussed in next chapter.

Example: Consider the definition of the `Attribute` linglet given in Section 5.7. The translational semantics of the linglet produces a `Column` node that has three parts: a table, a name of a column and an alias which is used for the result table. The `Attribute` linglet lacks sufficient parts for constructing the `Column` node as it has two parts: `tuple` and `name`. By default, we used the name of the attribute as an alias for the column. The alias is an essential part of a column in a `SELECT` clause, and thus an essential part of the translational semantics of an attribute. But, instead of embedding that into the translational semantics we created a new method `alias` which contained that logic and acts as a kind of virtual part. This facilitates the definition of the translational semantics of the `Atribute` linglet because no artificial distinction is made to retrieve the three parts which are necessary for constructing the `Column` node.

5.8.2 SP3-SP7 Acquisition of Multiple Inputs in Language Specifications

Recall from Section 4.3.2 that there are three steps required to acquire multiple inputs: identification, obtention and provision of the external information. The identification (SP3) and obtention (SP4-SP6) cannot be realized by the language specification because the language specification can only manipulate linglets i.e. compose and adapt them. This will be handled by interaction strategies. Provision of information (SP-7) to a linglet is supported by the language specification using two mechanisms: extension with behavior or extension with data.

SP7 - Extension with Behavior

The most common way for providing information in LTS is by extending the behavior of linglets. We already encountered this when we illustrated in the previous section how the compositionality conflict of the `Attribute` linglet can be solved. The `alias` getter method of the `Attribute` linglet is not a real part but a request for external information which has a default implementation.

SP7 - Extension with Data

Another mechanism for providing information in LTS is by extending linglets with new information. This kind of mechanism is more fragile because we need to ensure that the information is available when it is required by its translational semantics or by another linglet. Linglets should respect requirement R2c and only produce additional information which is an essential part of their translational semantics.

Extension with data is a necessary mechanism which is used for incremental computation. Such computations are used in the integration of nonlocal results (see Section 5.9.2 on page 207), but also for the computation of circularly defined external information. A circular definition of external information cannot be solved by merely computing the information using its parts because the parts themselves rely on the information that is currently being computed. Consider for example the following query:

```
 F = { x.firstname, x.name |
( ∃ p) ( rabbit(p) ∧ p.firstname = 'Bert' ∧ p.name = 'Fillemong' )
∨ ( ∃ p) ( rabbit(p) ∧ x.firstname = p.firstname
∧ x.lastname = p.name ∧ (∃ f) ( F(f) ∧ f.id = p.parent }
```

The query selects the rabbit breeding line starting with 'Bert Fillemong', by recursively selecting its children. Any property P one wishes to compute about the set F which depends on the definition of the set, depends on the P itself. In the case of circular definitions [WdMS+01], a fixed-point iteration is used for computing the final value of a derived piece of information. The algorithms assign an initial value and in each iteration that value is adjusted. When that value no longer changes, the final value is computed and the iteration is stopped. So, we need to assign and reassign values during the iterations. Once the computation is finished, these values can be accessed.

In order to implement such definition in LTS, we rely on the extensibility of the data of linglets and AST nodes. Information can be easily stored and retrieved using their corresponding getters and setters. Hence, linglets are made oblivious to the source of the information.

5.9 R3 - Multiple Results

As mentioned in Section 4.1.5, requirement R3 enforces that the translational semantics of a linglet can define effects on program fragments produced by the translational semantics of other linglets. The multiple results produced by the semantics of one linglet needs to be handled appropriately by other linglets such that the produced results get composed in the correct part of the target program.

Because linglets are isolated from one another, linglets can only *produce* multiple results.

5.9.1 R3 - Producing Multiple Results in Linglets

Linglets can produce multiple results which need to be integrated in different places in the target program. We have called these results nonlocal results. Nonlocal results are produced by hooking them to the local results (see Section 5.3.7) because only the local results are returned by the `generate` method.

The distinction between nonlocal and local results is important for the linglet using the produced program fragments. Because linglets use, combine and compose the program fragments so as to create their semantically equivalent target program fragment, and because the values preferably have to be semantically correct and definitely syntactically correct, linglets only accept certain kinds of AST nodes. In other words, nodes that are not non-acceptable are the ones do not yield a semantically and syntactically correct program fragment. Hence, by classifying the AST nodes that are not acceptable as nonlocals, a using linglet can ignore these nodes. The distinction is less important for the producing linglet, but as the distinction is very clear at the producing linglet it is often also made in that producing linglet: linglets that produce multiple values which cannot be combined into one expression, are also aware that a using linglet cannot accept all the various kinds of fragments. Therefore, linglets elect their primary result as their local result and their other results as their nonlocal results. If there is more than one local result or no distinction between a local and a nonlocal can be made, multiple results can also be returned by the `generate` method as a set, and all the results are then considered locals. When the distinction between locals and nonlocals is not so clear, a composition conflict can arise between the linglet using the multiple results and the linglet producing them. We refer the reader to Section 5.7 which detail how linglets and the language specification handle compositionality conflicts.

Example: In Section 5.7.2 we explained that the `Relation` linglet can be implemented in two ways. In this section, we discuss the implementation variant of this linglet which produces multiple results. The `Relation` linglet produces two results as shown below: a `True` node and an additional result being a `Table` node. The `Table` node is considered the nonlocal because the relation predicates are in fact operators or functions which return a boolean value. The `Table` node is therefore hooked into the `True` node as a nonlocal result. The role given to the `Table` node is the symbol `#table` (which is not be confused with the `#`-construct).

```
Linglet Relation {
   syntax {
      relation "(" tuple ")"
   }
```

```
   generate { | relation tuple res table |
(1)      relation := ast relation generate.
(2)      tuple := ast tuple generate.

(3)      res := #True{}.
(4)      table := #Table{ 'relation 'tuple }.
(5)      res nonlocals add: table role: #table.
(6)      res.
   }
}
```

Context Information

The integration of nonlocal results sometimes depends on the context i.e. the surrounding AST nodes (see SP12 in Section 4.3.3). In order to allow interaction strategies to operate on the context of a nonlocal, the initial context where a nonlocal is originated must be retained when it is produced. This initial context is essential to further process the nonlocals because it determines the integration semantics of the nonlocal (as we will further discuss in Section 5.9.2). A linglet must thus be able to indicate the initial context of nonlocals if required. The node where the nonlocal results are attached indicates the initial context of the nonlocal results in the target program.

Example: The integration of the nonlocal `Table` node produced by the `Relation` linglet (explained above) is not that straightforward. Some `Table` nonlocals may be discarded depending on the expression in which the relation is used. Consider the following query as an example why context is important:

```
{w.name | (∃ w) (¬ works_on(w) ∧ manager(w)...  }
```

Upon transforming this query, two nonlocal `Table` nodes are produced: `works_on w` and `manager w`. The integration of these tables in the `FROM` clause of the equivalent SQL query is not trivial. It is not possible to judge if a table is discarded only on the base of the source relation predicate. The whole expression must be analyzed. For this, the context of the nonlocal `works_on` table and the nonlocal `manager` within the expression is required. In this example, the `works_on` relation can be discarded in the equivalent SQL query because, according to the semantics of Tuple Calculus, quantified negated relations are expressions that always evaluate to true. This is due to the fact that there is always at least one node in the universe that does not belong to a given relation. Therefore, the table `works_on` should not be integrated while the table `manager` should be kept.

The algorithm for discarding and integrating `Table` nonlocals is explained in detail in the next chapter.

Identification of Nonlocals

As explained in Section 4.3.3 according to the three-party contracts, a producing linglet should also participate in the integration of its nonlocal results. In order to be able to integrate nonlocals, linglets need to identify them. To this end, role names are used. Roles decrease the coupling between the various language implementation concerns. In a language specification, nonlocal results are identified by their role names so as to specify their integration. By using role names, the language designer can abstract from the details of the results and thus respects the isolation of linglets. In other words, this identification is used to specify the integration of nonlocal results outside of the borders of linglets.

Example: The `Relation` linglet produced a nonlocal `Table` node. This nonlocal is named `table`. Via this name the nonlocal can be identified in order to encode its integration.

5.9.2 SP8-12 Handling Multiple Results in Language Specifications

Recall from Section 4.3.3 that handling multiple results consists of two steps: identification of the place where to integrate them and the integration itself. Due to requirement R3b a linglet is not allowed to explicitly consume results produced by other linglets unless these are essential for defining its proper translational semantics. In other words, other linglets cannot be charged with the additional responsibility of handling nonlocal results produced by other linglets. As the producing linglets, nor the consuming linglets can be made responsible for integrating the nonlocals, nor any another linglet can be made responsible for the integration, the only remaining option is to make the nonlocal results themselves responsible for finding the place where to integrate themselves. This leads to a specification of the algorithm for identification and integration in a single linglet i.e. the linglet that produces nonlocals.

In order to identify in which term the nonlocal result must become part of, one can either use the information encoded in the target terms themselves, or use the linglets that produce that target language term. We refer to the former as target-steered integration and to the latter as source-steered integration.

We first introduce the basic mechanism available in the language specification to add these responsibilities. The specific kind of behavior that implement these responsibilities is not hard-coded and dictated by the system but is contained in interaction strategies.

Basic Mechanisms

The basic mechanisms for adding the identification and integration responsibilities to nonlocals are:

1. **External identification of nonlocals** Nonlocals can be identified by their role which is given by the producing linglet when the method `nonlocal`*`role`*`: nonlocal` (see Section 5.4.2) is specialized.

2. **Extension of nonlocals with additional behavior** As detailed in Section 5.3.8, AST nodes can be extended with new method definitions. By encoding in these methods the behavior to identify the target program fragment in which to integrate a nonlocal, a nonlocal is made responsible for its integration.

   As the `nonlocal`*`role`*`: nonlocal` methods capture the initial context of nonlocals and their producing linglet, information from these locations can be easily accessed in the definition of the new methods. More precisely, by specializing nonlocals at run-time with new methods, the new methods can refer to variables in the current run-time environment (see Section 5.3.8). This is further illustrated in the example given below.

3. **Access to other ASTs** In order to identify and integrate nonlocals, we need to access the AST. As linglets are defined in isolation from one another, a linglet can only access it parts. So, only the methods which are added to the linglets in the language specification have access to the whole AST. However, access to sibling and parent AST nodes is required to be able to cover the whole tree. This access is provided by the reflective layer of LTS and interaction strategies which are discussed in the next chapter.

Example: Consider the query below that selects all employees who work more than 10 hours on 'ProjX'. The challenging part in this query is its formulation where two existential quantifiers appear right after each other and afterwards bounding the tuples. Although it is not formulated in a conventional way, it is mathematically sound.

```
{ e.name | employee(e) ∧ (∃ w) ( (∃ p) (project(p) ∧
works_on(w) ∧ w.ssn = e.ssn ∧ w.hours > 10 ∧
p.ssn = e.ssn ∧ p.project = 'ProjX' ))}
```

As each existential quantifier produces an `exists` SQL expression, two directly nested existential quantifiers raise an interesting integration problem. Figure 5.5 depicts the AST of the above source program and its equivalent target program.

As explained in the previous subsection, each `Relation` nodes produces a nonlocal `Table` node which is hooked to a local `True` node. In our example query, the three `Relation` nodes each produce a nonlocal `Table` and a local `True` node (see the nodes in typewriter font in Figure 5.5). The two directly nested existential quantifiers translate in a target program where two `Table` nodes are hooked as nonlocals in the expression of nested `Select` nodes. The integration of these nonlocals must ensure that the two `Table` nodes are integrated in the `FROM` clause

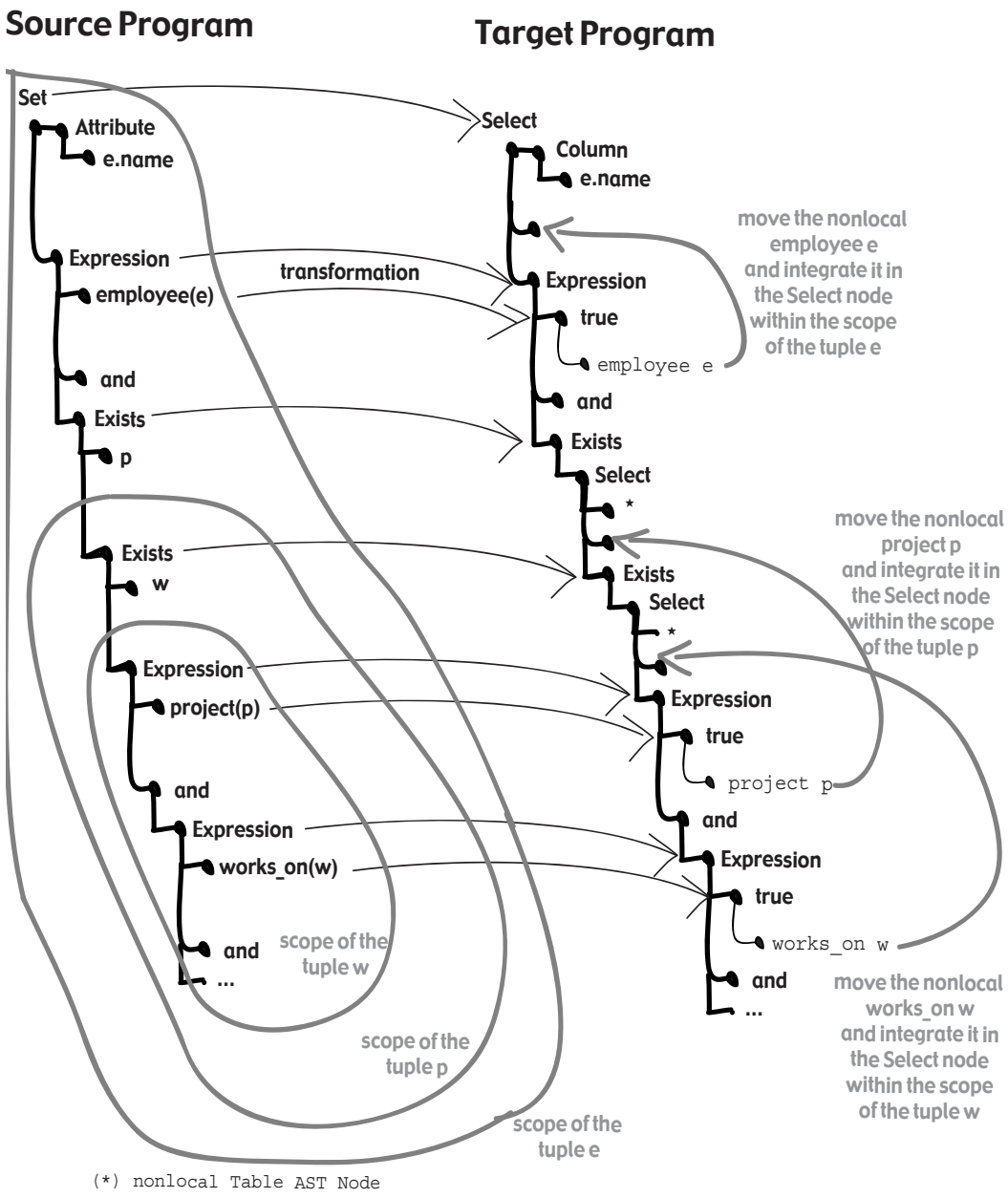

Figure 5.5: Source dependent integration of `Table` nodes.

of the correct `Select` node. More precisely, the integration must ensure that the nonlocal `Table` node representing the `project(p)` expression is integrated in the

inner most `Select`, that the nonlocal `Table` node representing the `works_on(w)` expression is integrated in the middle `Select` and that the nonlocal `Table` node representing the `employee(e)` expression is integrated in the outermost `Select`. In order to accomplish that, we need to identify the nonlocals and subsequently adapt them.

Below an excerpt of the language specification of T2SQL is given containing the integration of the nonlocal `Table` node. We identify the nonlocal `Table` with role `#table`, by overriding the method `nonlocaltable: nonlocal`. In that method, we extend the nonlocal with a new method `integratable: master` in line 4, which encodes the behavior computing whether a given AST node is a suitable node to integrate the nonlocal `Table`[18]. The `integrable: master` method works as follows. Recall that a relation is transformed into a `Table` node and that the tuple of a relation is an alias for the produced `Table` node in SQL. Tuples are scoped in Tuple Calculus by the quantifiers that declare them. In our example, the tuple `w` is scoped by the exists quantifier. A `Table` node is thus scoped according to the quantifiers in Tuple Calculus. In our example, the nonlocal table declaration `works_on w` is scoped by the exists quantifier $\exists$ `w`. Using the scope, we can thus decide where the table should be integrated or not. A nonlocal `Table` node is only integrated in a given `master` of the target AST when its source linglet is a scope (line 4) and declares the scope of its tuple (line 5). As such, nonlocal `Table` nodes are always integrated in those `Select` nodes which define the scope of the table.

The method `integratable: master` illustrates an interesting side effect of creating this behavior at run-time, since it uses the variable `tuple` which is defined in the enclosing method `nonlocaltable: nonlocal`. As such, the acquaintances of the `nonlocal` node in its method `integratable: master` are virtually extended with information from the `Relation` linglet.

```
Scope = Linglet
   tuple: {}

Set
   variable: Attribute.
   condition: Expression.
   extends: Scope.
   tuple: { ... }.

Exists
   extends: Scope.
   tuple: ID.
   condition: Expression.
```

[18]This method is part of a protocol prescribed by an interaction strategy to integrate nonlocals. More details are given in the following chapters.

```
Relation
(1)   nonlocaltable: nonlocal { | tuple |
(2)      tuple := ast tuple asString.
(3)      nonlocal integratable: master {
(4)         (master source linglet hasType: 'Scope') and: [
(5)            master source tuple = tuple
(6)         ].
(7)   }
}
```

SP8 - Identification via the Source Language Program

We have mentioned that the integration and identification is the responsibility of the nonlocal results themselves. A mechanism is necessary to access the AST node which produced that target node, so as to be able to guide the integration and identification using the source language in a method which is defined on a target node of the target language. The mechanism for gaining access to the source AST node is a pseudo part of the target AST node called `source`. This is a link between the target and source AST nodes provided by the reflective layer of LTS (explained in Section 6.1.5).

Example: In the previous example we illustrated that the integration of the nonlocal tables depends on the source language: the integration can only occur when the source AST node and source linglet of the target master define the scope of the table (see Figure 5.5). The source language provides a suitable abstraction level to encode the integration whereas SQL actually lacks an explicit scope declaration mechanism.

SP9 - Identification via the Target Language Program and SP12 - Context Information

Computing the decision whether or not to integrate nonlocal results based on the target language is more robust than the source language due to the reduced semantical complexity of its language constructs. In cases where the target language is less expressive than the source language, there are fewer language constructs for expressing the same observable semantics. As the semantics of the source language is defined in terms of the target language, the computations about the semantics of the source language expressed in terms of the target language in fact implicitly take into account the semantics of the source language constructs. As such, the semantics of the source language can be changed, which is more likely to happen than changes to definition and semantics of the target language, without having to change the computation of the integration logic. This is based on the Felleisen's conciseness conjecture [Fel91]. The conciseness conjecture states

No	query	Relations	
		works_on	manager
1	$e.name\|(\forall w)(works\_on(w) \wedge manager(w)...)$	FAIL	FAIL
2	$e.name\|(\forall w)(works\_on(w) \vee manager(w)...)$	FAIL	FAIL
3	$e.name\|(\exists w)(works\_on(w) \vee manager(w)...)$	OK	FAIL
4	$e.name\|(\exists w)(works\_on(w) \wedge manager(w)...)$	OK	FAIL
5	$e.name\|(\exists w)(\neg works\_on(w) \vee manager(w)...)$	DROPPED	OK
6	$e.name\|(\exists w)(\neg works\_on(w) \wedge manager(w)...)$	FAIL	OK
7	$e.name\|(\forall w)(\neg works\_on(w) \vee ...)$	OK	-
8	$e.name\|(\forall w)(\neg works\_on(w) \wedge ...)$	FAIL	-
9	$e.name\|(\exists w)(\neg works\_on(w) \vee ...)$	FAIL	-
10	$e.name\|(\exists w)(\neg works\_on(w) \wedge ...)$	FAIL	-

Table 5.1: Context-sensitive integration of nonlocal `Table`s produced by the relations in the listed queries.

that programs in a more expressive programming language that use the additional facilities in a sensible [19] manner, contain fewer programming patterns than equivalent programs in less expressive languages. What Felleisen actually is saying is that the closer the constructs in the programming language are to the tasks in the computation which needs to be expressed, the better. Hence, we can conclude that when the tasks of the computation are supported by constructs in the target language, the language constructs defined in the source language are not necessary to compute the integration. Moreover, excess logic (or code patterns) would have to be written to take the excess language constructs in the source language into account during integration.

Example: In the following example we show that the computation of the decision whether or not to integrate based on the target language is more robust and even less complicated compared to a source-steered integration.

Recall the integration of nonlocal `Table` nodes on page 202. We explained for a particular query that it is not possible to judge if a table is discarded only on the basis of the source relation predicate, but the context of the whole expression must be taken into account. Table 5.1 summarizes based on the context of queries indicate when the `Table` nodes produced by the relation should be integrated (OK), or when they cannot be integrated (DROPPED) or when the transformation process should fail (FAIL) because the query violates the semantics of the source language.
The table is constructed obeying the following rules:

- A positive relation fails when it is universally quantified, because not every tuple in the universe belongs to that relation.

[19] Sensible in this conjecture informally excludes the use of more expressive constructs for non-observable behavior.

- A relation fails integration when the same tuple must belong to two relations.
- A negated relation is acceptable when it is universally quantified in disjunction with an `or`, because every tuple in the universe can be considered part or no part of the relationship.
- A negated relation fails when it is universally quantified in conjunction with an `and`, because every tuple in the universe cannot be considered not to be part of the relationship.
- A negated relation fails when it is existentially quantified in conjunction with an `or`, when there is no other relationship left to define the tuple, it is dropped when another relationship is present. It can be dropped because there always exists a tuple in the universe which does not belong to that relation.

Consider the first line of Table 5.1. The relation `works_on` cannot be integrated because it is a positive relation in universally quantification of the tuple `w`. An universal quantification denotes that the given predicate must hold for every tuple in the universe called `w`. Obviously, the `works_on` relation predicate does not contain every tuple in the universe, so the transformation must fail. A similar reasoning can be formulated for each of the other listed queries.

The integration of the nonlocal `Table` nodes, based on Tuple Calculus, depends on five linglets which can be composed with a `Relation` linglet: `Exists`, `ForAll`, `And`, `Or`, `Not`. In order to compute when to discard, fail or integrate a `Table` node, one needs to take into account the composition of each of those linglets with other linglets and encode how they affect the integration. As we have shown, the rules to compute this decision are quite complex. More information on the drawbacks of a source-steered integration of nonlocal of this example are discussed in [CK07]. Clearly in this case, computing the decisions whether or not to integrate based on the source language is fragile as we explicitly rely on the existence of these linglets and their behavior. We therefore opt for a target-steered integration.

In SQL there are two linglets which can be composed and have an effect on a `Relation` linglet: `AND` and `NOT`. In order to compute when to discard, fail or integrate a `Table` node, one must only take into account those two linglets, and compute their effect of the linglet on the integration. The logic computing the integration is implemented in the `context_changed:node` method[20]. The logic encoded in the method is quite simple and is modeled after a state machine. Each encountered node changes the state of the integration. First two new variables

[20]This method is not part of LTS but is part of a protocol prescribed by an interaction strategy to integrate nonlocals. More details are given in the following chapters.

`discard` and `required` are stored in the linglet using their corresponding setters (see Section 5.8 for more information). When a `NOT` node is encountered, the discard and required flags are negated. When an `AND` node is encountered, the required flag is set to true.

We illustrate the integration of one nonlocal `Table` node with the example query number seven of Table 5.1. Figure 6.4 depicts the integration process for the nonlocal `Table` node.

```
Relation
   nonlocaltable: nonlocal {
      nonlocal discard: false.
      nonlocal required: true.
      nonlocal context_changed: node {
         node linglet type = 'NOT' ifTrue: [
             ast discard: ast discard not.
             ast required: ast required not.
          ] ifFalse: [
              node linglet type = 'AND' ifTrue: [
                 ast required: true.
              ].
          ].
      }
```

SP10 - Scheduling

In the previous examples, we showed that nonlocals are made responsible for their integration. The added logic to the nonlocals declaratively state when[21] to perform the integration, regardless of the fact whether other linglets producing the identified term or influencing the integration has already been executed or not.

SP11 - Integration

The integration of nonlocals in LTS is implemented as a three-party contract, carefully dividing the responsibilities (see Section 4.3.3). A three-party contract consists of (1) the linglet that produced the nonlocal and the nonlocal itself, (2) an external integration rule, and (3) the linglet that produced the program fragment in which the nonlocals must be integrated, and the produced the program fragment itself. The method `nonlocal`*`role`*`: nonlocal` and the specialization of nonlocal with new behavior clearly involves the first party. We also mentioned that the actually integration is specified by interaction strategies. These are infact the second party. The integration of a node in produced program fragments

[21] Nonlocals may also contain code to perform the integration (see next paragraph)

is specified by invoking its setters. Hence, by ensuring the consistency in their setters, those fragments actively participate in the integration of nonlocal results. This is the third party which is involved.

Example: Consider the `Exists` linglet shown below. In some situations the linglet does not produce an `exists` expression but a `join`[22]. However, a join operation of tables can produce duplicate results which are undesirable. To correct this, the `Exists` linglet produces a `Select` node with the `DISTINCT` keyword. This `Select` node must be integrated in (see `correspond: master` method) the `SELECT` clause defining the tuple query and it must (see `combine: master` method) add the `DISTINCT` keyword to the `Select` node upon integration. However, in case the tuple query is not a set but a bag, the `DISTINCT` keyword must not be integrated. This can be achieved because the integration is a three-party contract involving also the semantics of the `Bag` linglet. More precisely, the `Bag` linglet overrides the setter `distinct:` of its produced `Select` node to ensure that its `distinct` flag is not set.

```
Exists
   condition: BooleanExpression.
   tuple: ID.
   nonlocaldoubles: select {
      select correspond: master {
            master source linglet hasType: 'Query'
      }
      select combine: master {
            master distinct: ast distinct
      }
   }

Linglet Bag {
   syntax {
      "[" variable+ "|" condition "]"
   }
   generate { | variable condition query |
    variable := ast variable generate.
    condition := ast condition generate.
    query := #Select{ SELECT 'variable FROM dual WHERE 'condition }
    query distinct: value {
     (value ~~ #on) ifTrue: [
         previous distinct: value
     ]
    }
```

[22]When an attribute is part of the header of the set, then the translational semantics of a relation is not an `exists` but a `join`.

```
    }
}
```

5.10 Conclusion

The Linglet Transformation System (LTS) is a novel language development technique which fulfills the five requirements postulated in Chapter 4. In this chapter we introduced the core of LTS consisting of basic concerns and language specification concerns.

Languages in the Linglet Transformation System (LTS) are constructed from isolated language modules which we call *linglets*. Each linglet is a basic language concern. It thus defines the syntax and the translational semantics of a single language construct. The overall language semantics are defined by the language specification that composes linglets and establishes interactions among them.

The architectural style adopted by LTS is a prototype-based object-oriented organization. Each linglet represents a language construct and implements all operations that can be exercised on a program fragment that uses this construct. An instantiation of a linglet thus represents a particular program fragment and all cooperations and interactions among linglets are implemented by mere message passing between these instantiations.

The object-oriented prototype-based paradigm provides us with the ability to modify, compose and tailor linglets as well as instantiated linglets with concern-specific logic, which allows us to maintain the modularization of linglets. Even in the presence of more complex translational semantics, the semantics is appropriately localized in all the involved linglets. This high cohesion of a linglet and low coupling among linglets clearly shows that linglets maintain their strict separation.

Because linglets only define the semantics, they need to be completed in a language specification. Language specifications are incomplete at this point because the actual logic to establish the interactions cannot be expressed conveniently. More precisely, a linglet can only interact with its direct acquaintances. The interaction patterns and mechanisms are the subject of the next chapter.

Chapter 6

The MetaObject Protocol for LTS

The previous chapter introduced the kernel of LTS. The kernel is a transformation system in which each basic concern is captured in a modularized linglet.

Special purpose concerns are the mechanisms for implementing the interactions among linglets (see Section 4.2). Interactions occur when several linglets are combined which have more complex translational semantics, i.e. they do not compose, they require multiple inputs and/or produce multiple outputs. Translational semantics that does not compose need to be adapted so as to resolve the compositionality conflict. Translational semantics with multiple inputs require additional information besides the information contained in a linglet or translational semantics with multiple outputs require changes to the results produced by other linglets. We call this part of translational semantics in our modularization model the *effect* (see Definition 4.8).

In this chapter, we present a technique to implement the effects by using modularized implementations of special-purpose concerns, while maintaining the dominant decomposition of separate language concerns into linglets. So any logic specific to a linglet can be locally defined in that linglet. Any logic specific to a particular special-purpose concern can remain local to the special purpose concern.

The inability to effect the semantics at the level of basic concerns is rooted in the primitive communication mechanisms offered by the kernel of LTS. Recall that the kernel of LTS allows linglets to communicate directly with their immediate acquaintances only. The immediate acquaintances of a linglet consist of its children and its semantically equivalent target fragments. However, the interactions imposed by the complex transformations may stretch from neighboring linglets to distant linglets. As a result, many small interactions over these acquaintances, which we call intermediate linglets, are required in order to reach the more distant linglets and interact with them.

In Chapter 4 we discussed existing language development techniques (LDTs) (see Definition 2.2), each capturing a specific interaction strategy (e.g. forwarding,

structure-shy queries, symbol tables) that can be added so as to increase the degree of separation among basic language concerns. So in order to reduce the involvement of linglets, we will likewise also need interaction strategies.

In this chapter, we show that interaction strategies can vary. So it's unlikely that a fixed set of interaction strategies is sufficient, as will be apparent from the examples discussed in this chapter. The analysis of the interaction strategies identified in Chapter 4 already attests that they expose control over the execution of LDTs. The execution of the LTS is entirely expressed in terms of a set of interacting linglets. Hence, exposing control over the execution of LTS means exposing control over the execution of linglets. To this end, we equip LTS with a reflective layer for linglets so as to be able to construct new and tailored interaction strategies.

Reflection [Smi84] is the ability of a program to manipulate its state during its own execution. There are two aspects of such manipulation: introspection and intercession. Bobrow [BGW93] defines those terms as follows. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation. There are two types of computational reflection: structural and behavioral. Both are required in order to implement interaction strategies as they inspect and intercept linglets and change their semantics.

A metaobject protocol (MOP) organizes the reflective layer of a program such that it can be extended in an object-oriented style [Mae87, KRB91, Ste94]. For example in LTS, object-oriented concepts like dynamic dispatch, inheritance and information hiding are used at the meta-level to provide abstractions that hide implementation details of LTS while at the same time ensuring its extensibility. Moreover collaborations among objects naturally map to the concept of protocols and subprotocols in a MOP, resulting in an elegant system.

This chapter is divided into seven sections. Section 6.1 details the metaobject protocol designed to open up LTS. Afterwards, in Section 6.2, we explain how interaction strategies can in general be implemented using the MOP and sketch the implementation of the interaction strategies. The suitability, applicability and extensibility of the MOP is illustrated by implementing two interaction strategies in Section 6.3: one to retrieve information external to linglets, and one to resolve multiple outputs. We conclude our experiments with an advanced application of the MOP which is presented in Section 6.4, where we implement the languages to define linglets and language specifications in LTS itself. In Section 6.5, we revisit the tasks and challenges of the three kinds of special-purpose concerns which we identified in Chapter 4. We show how MOP can accommodate their needs. Before we conclude in Section 6.7, we reflect upon the essential and crucial qualities of the MOP to design and implement interaction strategies in Section 6.6.

6.1 MetaObject Protocol

The metaobject protocol reifies the fundamental concepts of LTS as first-class entities. As these entities represent linglets and encode their behavior, they are *metaobjects*. The protocol followed by the metaobjects to provide the behavior of linglets is called the *Linglet MetaObject Protocol* (LMOP). A static diagram of LMOP is shown in Figure 6.1.

The reified concepts in LTS are: `LS`, `Linglet` and `ASTNode` [1]. The `LS` metaobject reifies the language specifications, the `Linglet` metaobject reifies linglets and the `ASTNode` metaobject reifies the AST nodes.

Each metaobject reifies the structure and the behavior of its corresponding LTS concept. Intercession and introspection is only selectively applied to ensure the integrity of the system (see Section 6.1.6). For example, in Figure 6.1, the methods which can be intercepted are marked.

Interactions among the metaobjects define the basic behavior of LTS. They are described with sub-protocols i.e. smaller protocols that are a part of the overall protocol defined by LMOP. The kernel of LTS consists of isolated linglets and language specifications. Because the latter glues linglets, the behavior of a language implementation written in LTS is entirely defined and dictated by linglets. Furthermore, as linglets are defined in isolation, they have very limited communication abilities. Hence, there are only a few simple sub-protocols describing the basic interactions among LMOP metaobjects.

The remainder of this section consists of two parts: a description of the static part and a description of the dynamic part of LMOP. In the first three subsections we introduce and detail the static structural properties of the three reified concepts of LTS: language specifications, linglets and programs (linglet instances or AST nodes). The subsequent two sections each discuss the dynamics of the protocol for the various tasks performed by a linglet: the construction of its equivalent translational semantics by producing multiple target program fragments and the retrieval of the necessary context information. In these two sections, we include the necessary sub-protocols that describe the interactions and manipulations of metaobjects which define the basic behavior of LTS. The one but last section discusses how consistency of the base-level is retained given the intrusive access of the metaobject protocol. We conclude the discussion of the protocol with a summary.

6.1.1 Specifying Languages

Language specifications are reified by `LS` metaobjects. Through the metaobject, its three major parts are accessible: the header containing the name and the base language, the root linglets, and the test suites. Note that this information

[1] Not all the concepts encountered in LTS are reified. Syntactical parameters and methods are omitted as they behave passively in the semantics of LTS.

can only be introspected. More details on the reasons for this can be found in Section 6.1.6.

The source language specification can always be accessed through the global variable `LS`. The name of the source language and the name of the target language can be introspected respectively by the methods `NAME` and `BASE` defined in the `LS` metaobject.

Linglets

Root linglets can be introspected by the `LINGLETS` method defined in the `LS` metaobject. They are the entry points to linglet compositions and as such reifies the grammar.

Using the composition of linglets is rather tedious to look up linglets. To this end, a convenience method `LINGLET:aType` is provided by the `LS` metaobject in orrder to lookup linglets. This method traverses the composition of the linglets and looks up the linglets of a given type `aType`.

Parsing and Compilation

The language specification also provides methods `PARSE:aText` for parsing a complete program or a program fragment, and `COMPILE: anAST WITH: globals` for compiling it. The `anAST` parameter is the AST of a program written in the source language. The parameter `globals` (see also Section 6.1.5) is a dictionary of key-value pairs denoting the information which is external to the program. Global parameters are for example settings for the optimization level of a compiler, or search paths for looking up libraries, etc.

6.1.2 Specifying Linglets

The linglet metaobject reifies the type of a linglet, the syntax of its corresponding language construct, its translational semantics and its concern-specific methods.

The instance variables of a linglet metaobject cannot be assigned to after it has been created and initialized by LTS. This design choice is discussed in detail in Section 6.1.6.

Type

The most specific type of a linglet can be accessed through the method `TYPE`. Other types can be tested through the method `HASTYPE: aType` that determines whether the delegation chain of a linglet contains a linglet of the given type `aType`.

Syntactical Parameters

The syntactical definition of a linglet is not stored as a textual description but as a parser, modeled after parser combinators [SAA99], to retain both a high level and an executable specification. The syntactical definition refers to a set of syntactical parameters (see Section 5.3.3). Parameters can be inspected through the `PARAMETERS` method, yielding a list of symbols. The bindings of parameters can be retrieved by the method `PARAMETER: aName` but they cannot be altered (see Section 6.1.6).

Parameters can be single- or multivalued. This distinction is important in order to correctly compose AST nodes. The linglet metaobject provides the predicate `MULTIVALUED:aName` for determining whether a parameter with the given name `aName` is multivalued or not.

Semantical Methods

The translational semantics for a given AST node can be executed via the method `GENERATE: anAST`. Other methods can be reflectively invoked via the method `EXECUTE: aSignature ON: anAST WITH: anArgumentList`, where `aSignature` is the signature of the method to be executed, `anAST` is the receiver AST node, and `anArgumentList` the list of arguments for executing the method.

Besides reflectively executing methods, a limited form of introspection is available. One can only determine whether a linglet can respond to a method and obtain a list of abstract methods. The method `RESPONDSTO: aSignature` checks whether a linglet supports a message signature `aSignature`. The method `ABSTRACT` returns a list of signatures of abstract methods to which a linglet cannot respond. Methods themselves are not reified.

Semantical methods which are defined in a language specification or in a linglet behave slightly differently. Method declarations in a linglet must respect its isolation (see Section 6.4). Messages to retrieve acquaintances which are defined in the metaobject protocol or in one of its extensions may in principle *not* be called from within a method defined in a linglet. In order to enforce this, we distinguish between external method and internal methods. At run-time we check whether a certain method call is allowed from an internal method. This can be determined by the `EXTERNAL: aSignature` method.

6.1.3 Specifying Programs

Programs are represented with an AST, where each node is an instantiation of a linglet.

AST nodes

A new AST node representing a specific code fragment of a linglet can be created with the `NEW` message, which is defined on the `Linglet` metaobject.

AST nodes delegate to their linglets. The link from an AST node to its most specific linglet is exposed at the metalevel through the `LINGLET` method defined in each AST node. However, the delegation links between linglets are not accessible.

Parts

AST nodes form a tree navigable downwards from parent to child by storing their parts in a dictionary called `members`[2]. The `MEMBERS` method returns a list of symbols naming slots. Note that this function is not equivalent to the `PARAMETERS` of its linglet: some syntactical parameters may not have slots and some slots may not be syntactical parameters.

Each slot contains either a single AST node or a set of AST nodes. A set representation is necessary to avoid duplicate nodes in the AST. As nodes are changeable, duplicate nodes would cause unexpected changes when changing such nodes. When such behavior is indeed desired, it should be explicitly encoded.

Members can be accessed via the method `MEMBER: aName`. They can be updated, and new members can be added, via the method `MEMBER: aName ADD: aValue ON: index`. The index is an integer ranging from 1 to the number of nodes contained in a slot.

Both single and multivalued slots are thus manipulated using the same methods. Single-valued slots can only contain a single value and are thus always added at index 1. Because of the `ON` argument of the `MEMBER:ADD:ON:` method, the index of the value always needs to be specified. However, we add several convenience methods to modify the value of a single valued slot by `MEMBER:aMember PUT:aValue` or to add an element to a particular member by `MEMBER:aMember ADD:aValue`.

Parent

AST nodes form an upwards navigable tree through a parent link which is acessible via the `PARENT` method. Parent links among nodes are set by the kernel metaobjects.

6.1.4 Constructing Target Programs

Now that we have introduced and detailed the static structural properties of the three reified concepts of LTS, we proceed with the dynamics of the protocol for the various tasks performed by a linglet. In this and the next section, we include

[2]The members correspond to the slots or instance variables in object-oriented languages

the necessary sub-protocols that describe the interactions and manipulations of metaobjects which define the basic behavior of LTS.

All the sub-protocols of LTS are listed in Figure 6.2. We adopt the presentation style from Paepcke [Pae93]. The sub-protocol descriptions are depicted by a list of the various activities that must take place during the course of a sub-protocol. The activities correspond to a method call. A series of indented subactivities below each entry shows the necessary steps for accomplishing that entry. In other words, the subactivities below a method call are the method calls which need to be executed to implement that method call. Successive levels of indentations thus represent increasing levels of detail. The figures serve two purposes: giving a quick overview of a sub-protocol and showing the 'hooks' available for effecting changes.

In this section we explain the dynamics of the protocol involving the construction of equivalent translational semantics by producing (multiple) target program fragments.

#-Construct

The construction of a target program using the `#`-construct is syntactic sugar for a complex series of meta-level calls. Meta-level calls construct a target program by parsing an expression of the target language. Subsequently, each meta-variable `M`*`i`* is inserted into the correct location `K`*`i`* in the AST tree. The sub-protocol of the `#` construct is depicted in Figure 6.2.

Producing Multiple Results

Multiple results are produced by storing them in the `nonlocals` slot of an AST node. The base-level calls to retrieve and add nonlocals carry the same name as the meta calls, respectively `NONLOCALS` and `NONLOCAL: anAST ROLE: aRole` where `anAST` is the AST node which is added as nonlocal with the given role `aRole`. Alternatively, multiple results can simply be returned from methods, for example as sets. The resolution of multiple results, independent of how they are produced, is not further specified in LMOP but must be resolved by a suitable interaction strategy.

In the previous chapter, we have shown that nonlocals are directly retrievable via the interface of the linglet that produces them. This ensures that nonlocals are directly accessible although they can be hooked deep into the produced AST. For this, we define the nonlocal results sub-protocol which is depicted in Figure 6.2. The `T` and `S` name in the sub-protocol are both AST nodes, respectively denoting the target language AST node and the source or generating language AST node. The subprotocol invokes the method `NONLOCAL`*`role`*`: aNonlocal` on the generating linglet, where `aNonlocal` is a new nonlocal which has been created, and *`role`* is the `role` of the nonlocal (see Section 5.3.7). For example: the relation linglets

in T2SQL (see Section 5.9.1) produce a nonlocal *table*. So this nonlocal can be intercepted at the metalevel via the method NONLOCAL*table*:. This method is invoked upon the attachment of a nonlocal result via the NONLOCAL:ROLE: method.

6.1.5 Retrieving Information

In this section we discuss the dynamics of the protocol involving the retrieval of the necessary context information. The information sources of a linglet to compute context information can be subdivided into static and dynamic sources.

Static Information Sources

The static information sources for a linglet are its static acquaintances, i.e. its parts and its parent, and auxiliary methods in case the information is not locally available. Information requests to access their static information sources are implemented by method calls (see Section 5.8).

The subprotocol for requesting information is shown in Figure 6.2. Every information request starts out as a plain method call and is thus also treated as a plain method call. We first check whether a method is defined with that signature.

When no method is found and the signature has more than one argument, then the AST node cannot respond to the request and the UNKOWNREQUEST:aSignature WITH:anArgumentList is invoked. The default implementation throws an error, except for the following two cases:

- When the signature of a method request has one argument, and no method corresponds to that request, then that request is treated as a setter. Parts are set by the method MEMBER:ADD:ON:. As the AST nodes store their members in an open-ended form (see Section 5.3.2), any request for an unknown setter results in the creation of a new part. So when overriding the UNKOWNREQUEST:aSignature WITH:anArgumentList and invoking the overridden implementation, unknown setters can only be intercepted once.

- When the signature of a method request has no arguments, and no method corresponds to that request, then that request is treated as a getter. Parts are accessed by the method MEMBER:. As the AST nodes store their members in an open-ended form (see Section 5.3.2), any request for an unknown getter results in the creation of a set such that multiple AST nodes can be added. Consider for example the following code excerpt:

```
ast unkownpart add: otherast.
```

It gets an unknown part `unkown` which returns a set, and to this set a node `otherast` is added.

So when overriding the `UNKOWNREQUEST:aSignature WITH:anArgumentList` and invoking the overridden implementation, unknown getters can only be intercepted once.

Globals

Another static source of information for linglets is the access to information which is external to a language. We call this global information. Global information can only be accessed in a language specification, in order to preserve the modularization of linglets as linglets are not designed to operate in a specific language but should be usable in many languages. Global information can be accessed via the method `GLOBAL: aKey`.

Source

LTS provides the source link as a special static acquaintance to linglets in addition to its parent and its parts. The source of an AST node points to the AST node which produced it. This link is automatically set by the system and can only be accessed through the `SOURCE` part.

The source acquaintance plays a crucial role for debugging, tracking and interaction strategies which use the source language as an abstraction level for the target language. The latter are for example used to implement source-steered integration of nonlocal results (see Section 4.3, Section 5.9.2 and Section 7.5.10).

Caller

Besides static information sources, a linglet also has a dynamic information source. There is currently only one dynamic acquaintance offered by linglets and that is the caller of a semantic method of a linglet. A linglet can thus reflect on the dynamic chain of semantic method calls issued by previous linglets through the `CALLER` method.

The reason for adding this is that many interaction strategies implicitly use the caller. Consider, for example, the threading of a symbol table through an interpreter. Each time a function is invoked, the current symbol table is passed from the caller to the callee. As it is our intention to provide an open language development technique, where interaction strategies can be implemented as custom linglets, LTS explicitly provides access to this information.

6.1.6 Consistency

Steyaert [Ste94] warns us that an open implementation does not define a single system but an entire design space of (related) systems. Merely opening up a system's implementation gives no guarantee that the so created design space is coherent. Furthermore, he argues that it is therefore important to identify the constraints surrounding the language concepts that are made explicit.

When adding a meta-layer to LTS, the consistency of linglets must not be compromised by the control over the behavior and over the structure of a language implementation obtained through the meta-layer. Intercession can easily disrupt the basic assumptions and behavior of a system. The goal in designing LMOP was to expose the behavior of the transformation process so as to be able to extend LTS, while limiting the risk of accidentally breaking and corrupting LTS.

The following design decisions are also graphically depicted in Figure 6.1.

Stable Information

Intercession of structural information which needs to remain stable throughout the execution of a transformation is prohibited. The specification of source and target language are the prime examples of such information. We call this *grammar stability*. Grammar stability ensures that interaction strategies relying on the grammar behave consistently during the transformation.

Consistent Information

Intercession of structural information which needs to remain consistent is carefully controlled. AST nodes have two kinds of static acquaintances: their parts and their parent. In order to enforce local consistency, intercession of these acquaintances must be managed.

AST nodes form a tree through a parent link. Intercession of the parent is not possible. The parent link is maintained by the metalevel and is implicitly changed when parts are added or replaced. Hence, the local consistency of the containing node is ensured.

Linglets can enforce local consistency by providing additional methods for guarding the setters of the parts of an AST. LTS ensures this consistency by reversing the implements-by relationship between the base and the meta level: by enforces that all changes to parts originating both from the base and the metalevel go through the guarded setter methods of the base level.

Hidden Implementation Details

Implementation details are hidden by limiting intercession of behavior. The prototype-based character of LTS treats linglets and their AST nodes as objects whose behavior can be progressively specialized. Linglets and AST nodes

are in fact a series of objects that delegate to each other. This delegation chain is hidden from both the base and the metalevel to preserve the identity and to centralize data in order to maintain update consistency of the AST nodes during the transformation process (see Section 5.3.8). Therefore metaobjects are equipped with a pair of methods that abstract over the delegation chain.

Metaobjects provide methods that allow interaction strategies to use the information provided by the delegation chain. The `RESPONDSTO:` method determines if an AST or linglet can respond to a particular message call. Another example is the `HASTYPE:` method to determine whether an AST or linglet is of a particular type. In both methods, the delegation chain is taken into account.

Completing Behavior

Only some of the methods at the metalevel can be overridden. They are marked by a double star in Figure 6.1. Examples of non-overridable methods are `HASTYPE:` (see Section 6.1.2) and `NONLOCAL:ROLE:` (see Section 6.1.4).

Bulk of the structural information cannot be altered because there are simply no setter methods defined in the metaobject protocol. However, as all structural information is accessible by inspectors, these accessors have to be made non-overridable as well. Examples of such information are the `CALLER` (see Section 6.1.5) and the `PARENT` (see Section 6.1.4) getters.

Intercession in behavioral information is designed to complete the basic meta behavior of a linglet in case of: introspection and intercession of members, the execution of the translational semantics and the execution of methods. These methods are always executed first by the kernel metaobjects, afterwards metaobject extensions can respond to these calls. There are two such methods in the metaobject protocol, the method `MEMBER:` (see Section 6.1.4) and the method `RESPONDSTO:` (see Section 6.1.2). When a member is retrieved by calling the method `MEMBER:`, and a value for that member has been found, that value is returned regardless of the return value of the overridding `MEMBER:` methods. The same is true for the return value of the method `RESPONDSTO:`. Only in case there is no method or part which can respond to the given signature, can the overriding `RESPONDSTO:` methods refine the return value to true. As a result, members and functionality cannot be hidden.

These above methods have a similar feel to the inheritance scheme used in Beta [BC90], as inheritance in Beta is designed to provide security from replacement of a method by a completely different method. However, only the mentioned methods defined on the kernel metaobjects have this inheritance scheme. Extensions of these metaobjects use an inheritance scheme like in the kernel of LTS (see Section 5.3.4).

The other overridable methods (see Figure 6.1) can completely override those of the kernel metaobjects. As such, a developer can add logic before and after the execution of those methods and even prevent their execution if required.

6.1.7 Putting It All Together

Table 6.1 lists the meta-level calls which correspond to the various base-level calls. These meta-level calls are the primary means for interaction strategies to influence the behavior of the linglet.

6.2 Interaction Strategies

Interaction Strategies use the meta level of LTS in order to extend linglets with additional functionality so as to facilitate the implementation of the more complex and challenging translational semantics. In this section, we explain how interaction strategies can be implemented in LTS using LMOP.

In the first section, we explain how interaction strategies fit in the two level architecture of LTS i.e. how interaction strategies fit in its base level and its meta level.

6.2.1 Situating Interaction Strategies in LTS

Interaction strategies are implemented as collaborations between linglets and need to exercise control over the behavior of linglets. Hence, interaction strategies operate on the base level and on the meta level. We further observe that the base-level parts and the meta-level parts of an interaction strategy tightly cooperate. A meta-level call can invoke base-level calls and a base-level call can invoke meta-level calls. The underlying reason for this lies with the generic nature of interaction strategies. Consider for example interaction strategies for handling multiple results. Such interaction strategies can encode some generic behavior to traverse and direct the integration, but the actual logic deciding whether a particular AST node should be integrated and how, needs to be specialized by that specific AST node. As this logic solely deals in terms of specific AST nodes, that logic is best written at the base-level.

In LTS the base and the meta level are not stratified. As a result, an interaction strategy can be implemented as a single linglet extension.

6.2.2 Implementing Interaction Strategies

LMOP only exposes the internal structure and behavior of LTS. This fixed protocol offers basic abstractions and basic functionality necessary to implement custom interaction strategies. Through extension of the predefined metaobjects with new metaobjects, interaction strategies can be added to LTS (see Figure 6.3).

A new metaobject defines all the additional methods required for the protocol of a new interaction strategy. The methods contain parts of the protocol which are common to all linglets that use the interaction strategy (L2 of Figure 6.3).

BASE LAYER	META LAYER
`C1 := ast param1 generate.`	`C1 := ast EXECUTE: #param1 WITH: nil` `ast GENERATE: WITH: nil`

`C2 := C1 info.`	`C2 EXECUTE: #info WITH: nil`
	`MEMBER: #info`
	`UNKOWNREQUEST: #info WITH: nil`

`targetprog = #L{}.`	`L := LS BASE LINGLET: #L.` `targetprog := L NEW.`

`targetprog part1: C1.`	`targetprog EXECUTE: #part1 WITH: C1`
	`targetprog MEMBER: #part1 PUT: C1.`
	`ast part1: C1`

`targetprog part2 add: C3.`	`targetprog EXECUTE: #part2 WITH: C3`
	`targetprog MEMBER: #part2 ADD: C3.`
	`ast part2: C3`

`ti =#type target code 'C1 .`	`ti = (LS BASE PARSE:(" target code Ki::",`
	`C1 TYPE, " ") LINGLET: type)`
	`K1 PARENT MEMBER: #m PUT: C1 ON: Ki`

`targetprog nonlocals add: ti role: aRolei.`	`targetprog NONLOCALS ADD: ti ROLE: aRolei`
	`targetprog NONLOCAL: ti ROLE: role`
	`ti ROLE: rolei`
	`ast LINGLET NONLOCAL`*`aRolei`*`: ti`

Table 6.1: Overview of the base-level calls corresponding to the respective meta-level calls.

If there is no common part for an activity of a protocol, the method corresponding to the activity is an abstract method (starred methods in Figure 6.3). The metaobject also contains a number of overridden or refined methods that hook the extension into the execution of linglets and thus into LTS (`methodX` in L2 of Figure 6.3). This way the protocol of an interaction strategy is clearly specified for all involved parties and the discrete metaobjects that define interaction strategies can be reused across multiple language implementations. The parts of the metaobject protocol that are not common are left to the `linglet` (`methodX` in L1 of Figure 6.3) or to the `ASTNode` (`methodX` in L0 of Figure 6.3) further to be specified. A concrete example of interaction strategy deployment specialized down to the level of the linglets is given is depicted in Figure 7.4 in Section 7.4.10.

Linglets and interaction strategies are defined in isolation with respect to the rest of a language. Only in a language specification, linglets are finally specialized with interaction strategies. As interaction strategies sometimes need to impact all linglets in a language specification, the root linglet of the delegation chain called `Linglet` (see Section 5.4.2) can be specialized. An example of such a deployment is shown in Section 6.3.1.

6.3 Experiments with Interaction Strategies

The major difference between LTS and other transformation systems is its ability to customize and change the transformation system so as to incorporate new interaction strategies through its LMOP. By discussing a number of concrete interaction strategies, we illustrate the extensibility of the LTS system but also the impact of that extensibility on the design of interaction strategies.

In the following two sections we conduct two experiments in which we presented a family of interaction strategies for retrieving context information and for declaring and specifying the scattering of code fragments. The former is a variation of an existing interaction strategy from another language development technique, the latter is an entirely new interaction strategy. Both interaction strategies are used in our case study where we validate this dissertation (see Section 7.4.10 and Section 7.5.10).

6.3.1 Existing Interaction Strategies: Structure-shy Queries

In this section, we present an interaction strategy for completing linglets with translational semantics that require external information. The interaction strategy implements structure-shy queries similar to the ones found in XSLT [Lai] via a LMOP extension. We refer to this interaction strategy as the SSQ strategy, which stands for Structure-Shy Queries. SSQ enables us to look up information that resides in other linglets by specifying a path to another AST node in the source or target AST using basic operations like descendant to find a particular

AST node in a given subtree, or ancestor to find a particular AST node containing a given node. Paths in XSLT do not specify the actual path in the AST that needs to be followed for obtaining information. We introduce the protocol gradually, starting with a very basic protocol followed by adding a number of extensions. As such, we illustrate the flexibility for adapting and configuring interaction strategies using several layers of extensions.

Interaction Strategy Protocol

The basic simple SSQ protocol (see below) consists of three basic structure-shy operations: *descendant*, *ancestor* and *descendants*.

Each operation is implemented with two mutually recursive methods. *Descendant* is implemented with `DESCENDANT:aType` and `DESCENDANTWITH:aType`, *ancestor* with `ANCESTOR:aType` and `ANCESTORWITH:aType`, and *descendants* with `DESCENDANTS:aType` and `DESCENDANTSWITH:aType`. `DESCENDANT:aType` finds a node whose linglet is of a certain type from the subnodes of a current node, which means that the current node is excluded in the search. `DESCENDANTWITH:aType` does the same but includes the current node in the search. Likewise, `ANCESTOR:aType` finds a node whose linglet is of a certain type from the parent nodes of a current node, and `ANCESTORWITH:aType` does the same including the current node. `DESCENDANTS:aType` and `DESCENDANTSWITH:aType` are similar to `DESCENDANT:aType` and `DESCENDANTWITH:aType` but collect all nodes whose linglet is of certain type instead of returning just one matching node.

Protocol - Basic simple SSQ:

```
DESCENDANT: aType
      DESCENDANTWITH: aType
ANCESTOR: aType
      ANCESTORWITH: aType
DESCENDANTS: aType
      DESCENDANTSWITH: aType
```

Illustration

Querying the source or target AST is not only used for completing the translational semantics requesting multiple inputs. External information is also necessary in order to compute other translational kinds of completions or concern-specific logic. As an illustration of the SSQ strategy, we complete the resolution of the compositionality conflict introduced in Section 5.7.2 on page 197. A composition conflict occurs between the `attribute` linglet, used as a variable in the

header, and the ID linglets in case this attribute participates in an equality relation.

```
(1)   { t.family |
(2)     employee(t)  ∧
(3)     t.family = t.lastname  ∧
(4)     t.wage = 50.000  ∧
(5)   ( ∃  w)( works_on(w) ∧ w.ssn = t.ssn ∧ w.project = 'X' ) }
```

In the above T2SQL query (see Section 5.1), the name, tuple and alias parts of the attribute node t.family are composed with the AST nodes family, t and family respectively. However, because t.family is bound with the attribute t.lastname, the composition of the Attribute node is incorrect. In the binding of the attribute t.family with the attribute t.lastname, the latter contains the actual name and tuple, the former denotes the alias. So actually, the name, tuple and alias parts of the attribute node should be composed with the AST nodes lastname, t and family respectively.

In order to resolve the conflict, the accessors of the alias, the tuple, and the name, of the Attribute linglet are specialized. The two accessors invoke an auxiliary method findAssignedAttribute:name: in order to retrieve the binding of the equality relation which is a BinaryOperation with an Equal operation operating on two Attributes. The binding is retrieved by invoking the descendant method of the SSQ strategy.

```
Attribute
  ...
  findAssignedAttribute: aTuple name: aName {
    bin := (ast descendant: 'BinaryOperation') detect: [
       bin operator hasType: 'Equals'
       and: [ bin left linglet hasType: 'Attribute' ]
       and: [ bin right linglet hasType: 'Attribute' ]
       and: [ bin left tuple asString = aTuple ]
       and: [ bin left name asString = aName ]  ].
    (bin == nil) ifTrue:[ nil ] ifFalse:[ bin right ].
   }
```

In the code above we assume that the attribute of a tuple in a header occurs on the left hand side of the binding. In general this is not always the case. The above can easily be adjusted so as to deal with other cases.

Implementation

There are two ways for implementing the SSQ strategy: iteratively or recursively. In an iterative implementation, the logic of a basic operation is contained in a single linglet metaobject. Descending an AST, for example, can be performed in a single method. In a recursive implementation, one meta-linglet relies on the meta-linglet of its neighbours. In order to descend an AST, linglet metaobjects recursively continue the traversal in its neighbours. At first sight, the iterative implementation may be more favorable because of its limited impact on the implementation i.e. only one linglet must be equipped with this interaction strategy. However, in complex queries (for example the query on page 233) where several basic operations are concatenated, every AST can be the starting point of a subquery. Hence, every linglet must be equipped with this interaction strategy. Therefore, we opt for a recursive implementation.

The SSQ strategy is implemented in a separate linglet called `SSQ`. The first and second method retrieve the descendant whose linglet matches the given type `aType`. They are mutually recursive methods. The method `descendant:` iterates over the members of an AST node and subsequently calls the method `descendantwith:`. This method checks whether the linget of the current AST node matches the given type. If so, a descendant has been found. If not, the search for a descendant is continued.

```
Linglet SSQ {
  descendant: aType {
     ret:= nil.
     ast members detect: [ :member | | ret |
        (ast member: member) detect: [ el |
             (ret:= el descendantwith: aType ) ~~ nil  ] ~~ nil.
        ret
  }
  descendantwith: aType {
   (ast linglet hasType: aType)  ifTrue: [ ast ] ifFalse: [
             ast descendant: aType.
   ]
  }
}
```

Note that in this discussion, we only focus on a single basic operation, namely `descendant:`. The other operations of the interaction strategy are similar.

The SSQ strategy extends the metaobject protocol of LTS. It provides new methods allowing communication with more distant linglets by introspecting the structural information of an AST node.

A typical deployment of the SSQ strategy is shown in the following language specification:

```
T2SQL
base SQL

Linglet=SSQ
   extends: Linglet.
```

The interaction strategy is deployed such that every linglet and their instances are equipped with this protocol. To this end, the `Linglet`[3] linglet is aliased to the `SSQ` Linglet, which in return extends the default `Linglet`.

6.3.2 Adjustment of Existing Interaction Strategies

There are several opportunities to further extend the SSQ strategy. It is not our intention to list them all but rather to illustrate that interaction strategies can extend existing interaction strategies and that multiple variations of the same interaction strategy can be used, therefore we limit this discussion to three sub-concerns. In this section we present two extensions. The first extension is to allow nested queries i.e. descend and ascent based on a condition rather and on the type of the node to be found. The second extension is to limit the scope of a query i.e. descend or ascent a limited part of the tree rather than traversing the whole tree. We start by revisiting the basic strategy to pave the way for future extensions.

Basic Queries

Before we can start discussing the various extensions we need to revise the implementation of SSQ described up till this point in the dissertation. The three sub-concerns of the SSQ strategy we cover in this section are: the parameters influencing the behavior of the query, the tests determining whether an AST node satisfies the query, and the execution of the query. The new protocol reflecting these three sub-concerns is shown below:

Protocol - SSQ Basic:

```
DESCENDANT: aType
   _DESCENDANT: aType IN: args
   _DESCENDANTWITHOUT: args
      _DESCENDANTWITH: args
         _TESTDESCENDANT: args
```

The basic strategy is implemented in the linglet `SSQBasic`, which is shown below. The implementation is a straightforward mapping of the protocol.

[3]The `Linglet` linglet is the root of the delegation chain (see Section 5.4.2)

```
Linglet SSQBasic {
  descendant: aType {
     ast _descendantwithout:
        (ast _descendant: aType in: Dictionary new)
  }
  _descendant: aType in: args {
     args at: #type put: aType.
     args
  }
  _testdescendant: args {
     (arg includesKey: #type) ifTrue: [ | type |
        type := (arg at: #type).
        (ast linglet hasType: type)
     ] ifFalse: [ true ]
  }
  _descendantwithout: args {
     ret:= nil.
     ast members detect: [ :member |
        (ast member: member) detect: [ el |
           (ret:= el _descendantwith: args ) ~~ nil  ] ~~ nil
     ret.
  }
  _descendantwith: args {
     (_testdescendant:args) ifTrue: [ ast ] ifFalse: [
        ast _descendantwithout:args
     ]
  }
}
```

The parameters of a query are stored in a `Dictionary` called `args`. The method `_DESCENDANT: aType` hides the dictionary from the language implementor using this interaction strategy. The method appropriately initializes the dictionary when a descendant of a particular type is searched. The test determining whether or not an AST satisfies a query is performed in `_TESTDESCENDANT:args`. The method returns true on a positive match. The method retrieves the argument `#type` from the dictionary and checks whether the current AST is of a particular type. The execution of the traversal is captured in two mutually recursive methods `_DESCENDANTWITHOUT:args` and `_DESCENDANTWITH:args`. The method `_DESCENDANTWITH:` iterates over the members of an AST node and subsequently calls the method `_DESCENDANTWITHOUT:args`. This method checks whether the current AST satisfies the query by calling the `_TESTDESCENDANT:args` method. If this returns true, a descendant has been found. If not, the search for a descendant is continued by calling the `_DESCENDANTWITH:args` method.

The methods in the protocol that start with an underscore are included in the protocol to provide more possibilities for customizing it.

Extension: Nested Queries

SSQ is modeled after the operators found in XPath [BB02]. The basic strategy presented above, however, oversimplifies XPath. XPath also supports more general conditions in the form of predicates. These predicates state which nodes are to be selected or found when the AST is traversed. They are defined in terms of basic operations and functions. As such, queries can be nested. In order to allow the same kind of expressiveness as in XPath, we implement an extension of the SSQBasic strategy called SSQCondition. The protocol of the interaction strategy is depicted below.

Protocol - Nested Queries:

```
DESCENDANTIF: aCondition
      _DESCENDANTIF: args IN: args
      _DESCENDANTWITHOUT: args
```

The protocol has been extended with a `DESCENDANTIF:aCondition` method taking a condition instead of a type. As the extension only influences the test determining whether or not an AST node satisfies the query, only the method `_TESTDESCENDANT:args` is overridden. An AST node satisfies the query if the node adheres to the condition and if the previous `_TESTDESCENDANT:args` method evaluates to true.

```
Linglet SSQCondition {
  descendantif: condition {
   ast _descendantwithout: (ast _descendantif: condition
                                in: Dictionary new)
  }
  _descendantif: condition in: args {
   args at: #condition put:  condition.
   args
  }
  _testdescendant: args {
   (args includesKey: #condition) ifTrue: [ | block |
      block := (arg at:#condition).
      (block value:ast)  ifTrue: [ previous _testdescendant:args ]
                         ifFalse: [  false ]
   ] ifFalse: [ previous _testdescendant:args ]
  }
}
```

Extension: Bounded Queries

The second extension to SSQ illustrates that existing interaction strategies can be extended without holding true to their original or initial conception. In this example, we extend the SSQBasic strategy with the ability to limit a query to a certain scope. This interaction strategy is called SSQScope. Note that scoping is not explicitly[4] supported by XPath.

Scoped queries are called for in the implementation of the T2SQL language (see Chapter 4). In multilingual databases, the column names containing multilingual information are duplicated for each language and postfixed with a language identifier. So for an attribute `foo` in a multilingual database supporting Dutch and French, two columns `foo_nl` and `foo_fr` are created. Queries executed against a multilingual database must return only one of the values depending on the context. Queries in T2SQL are formulated in terms of attributes and not in terms of columns. Hence, the actual column names that must be used in case of a multilingual attribute are computed by the T2SQL compiler. Consider the following query, where the `project` attribute of a `works_on` relation is a multilingual field which is retrieved by the attribute `t.project` in the header of the set.

```
(1)   { t.family, t.project |
(2)     employee(t)  ∧
(3)     t.family = t.lastname  ∧
(4)     t.wage = 50.000  ∧
(5)     (∃  w)( works_on(w) ∧ w.ssn = t.ssn ∧ w.project = t.project)
(6)   }
```

In order to resolve the actual name of the attribute `t.project` in the header of the set, we first need to lookup its binding to the attribute `w.project`. Afterwards, the relation `works_on` of the tuple `w` is searched within the scope of the declaration of that tuple `w`. The query in the second step is bound to the scope of the declaration so as to prevent entering subscopes that shadow the declaration. Once the relation `works_on` of the bounded attribute `t.project` is found, one can determine whether the attribute `project` is multilingual by consulting the database schema and compute the correct column name. The database schema is an example of additional information which is external to the compilation (see SP4 in Section 4.3).

In the query below, we present the snapshot of the code that resolves multilingual fields. It queries for the relation within the declaration scope of a tuple. First, the assigned attribute is computed using the method `findAssignedAttribute:name:`, which is defined in Section 6.3.1. Second, the declaration of the tuple of the binding is looked up. Last, the relation of that tuple is searched within the scope of the current declaration.

[4]Scoping is implicitly supported through the language which XPath is embedded in.

```
Attribute
  ...
  findRelation: aTuple name: aName {
    bin := findAssignedAttribute: aTuple name: aName.

    (bin == nil) ifTrue:[ nil ] ifFalse:[ | attribute |
       attribute := bin right.
       declaration := attribute ancestorif:
              [ scope | (scope hasType: 'Scope')
                        and: [ scope tuple asString
                               = attribute tuple asString ] ].
       declaration descendantif: [ rel  |
           (rel hasType: 'Relation')
           and: [ rel tuple asString = aTuple  ]
       ] scope: [ scope | (scope hasType: 'Scope')
                  and: [ scope tuple = attribute tuple asString  ] ].
    ].
  }
```

The protocol of the SSQScope strategy consists of a single method:

Protocol - Scoped Queries:

```
    _DESCENDANTSCOPE: aCondition IN: args
```

The interaction strategy is implemented by the linglet SSQScope. This interaction strategy overrides the _DESCENDANTWITH: method. It first checks whether the current AST node exceeds the scope of the descent. If so, the descent is aborted, otherwise the descent is continued by invoking the previous definition of the _DESCENDANTWITH: method.

```
SSQScope {
  _descendantscope: scope in: args {
     args at: #scope put:  scope.
     args
  }
  _descendantwith: args {
     (args includesKey: #scope) ifTrue: [ | block |
         block := (arg at:#scope)
         (block value:ast)  ifTrue: [ previous _descendantwith:args ]
                            ifFalse: [
                              nil
                            ]
```

```
      ] ifFalse: [ previous _descendantwith:args ]
  }
}
```

Notice that all methods are hooks (prefixed with an underscore). This indicates that this interaction strategy must be further completed with a method that offers a convenient way to use this extension.

In the language specification below, `Linglet` is extended with one interaction strategy and two extensions: first with `SSQ`, subsequently with `SSQCondition` and finally with `SSQScope`. The latter linglet is completed with the method `descendantif:scope:`. This method brings together the functionality of the `SSQCondition` and `SSQScope` linglet. It launches a scoped descent query in search of an AST node adhering to a particular condition.

```
SSQCondition
  extends: SSQ.

Linglet= SSQScope
  extends: SSQCondition.
  descendantif: condition scope: scope {
   args := Dictionary new.
   args := ast _descendantscope: scope in: args.
   args := ast _descendantif: condition in: args.
   ast _descendantwithout: args
  }.
```

6.3.3 New Interaction Strategies for Multiple Results

In this section, we present a new interaction strategy to complete linglets producing multiple results. We refer to this interaction strategy as the INR strategy, which stands for Incremental Nonlocal Results. The interaction strategy identifies where the nonlocals should be integrated and performs their integration.

The INR strategy is interesting for several reasons. First, the INR strategy is a pioneering example of the introduction of successful techniques from compositional generators into transformational systems. Second, this interaction strategy invasively changes the behavior of LTS. Third, to the best of our knowledge this interaction strategy is the only existing mechanism that effectively modularizes the specification to scatter multiple results in a target program.

The INR Strategy

The INR strategy is a generic interaction strategy that manages the lifecyle of nonlocals starting from the point where they are produced to the point where they

are integrated in a target program AST. Like any other interaction strategy, INR operates on and is defined at the meta level of LTS. The INR strategy can thus reason about the target language and its implementation, but it is not designed for a specific language implementation.

The INR strategy offers a language independent implementation of the relocation and the integration of nonlocals. Relocation is a step-by-step process moving nonlocals node by node upwards in the AST. Integration is a semi-automatic process. Based on the grammar of the target language, INR decides whether a nonlocal can be integrated in a given target node or not.

The INR Strategy Protocol

The INR strategy handles each nonlocal throughout the execution of a transformation. The location in the target program AST in which to integrate nonlocals is computed incrementally by using a fixed point algorithm. The fixed point is the point where nonlocals are no longer moved and can be successfully integrated. When one fixed point exists the transformation process succeeds, when no fixed point exists it fails. The case where many fixed points are found is discussed in Section 6.3.3 on page 244. The incremental nature of the INR strategy allows nonlocals to be processed even if the actual location in the target program in which to integrate them has not been computed yet. The scheduling of the production of nonlocals, the creation of the target location and the processing of the nonlocals is handled by the protocol.

The INR strategy establishes a rather complex collaboration between three roles: the subject, the master and the consumer. The nonlocal plays the *subject* role. The node in which to integrate the nonlocal plays the *consumer* role. The intermediate nodes along the path from the nonlocal to its consumer play the *master* role.

Nonlocals are explicitly bound by the interaction strategy, as nonlocals are made responsible for integrating themselves. In other words, the meta level of nonlocals is equipped with the INR strategy. The consumer and master roles cannot explicitly be bound to certain nodes, as the nodes which consume nonlocals and the intermediate nodes in between are to be computed by the INR. Explicitly binding them would mean that we can statically identify, without any computation, where nonlocals should be integrated. Clearly that is not the case.

The protocol of the INR strategy is depicted below. The lower case names indicate the roles used in the various activities of the protocol.

Protocol - INR Strategy:

```
aggregate MEMBER: aName PUT: part
   subject INTEGRABLE: master
   subject RELOCATETO: aggregate FROM: part
```

```
subject INTEGRATE: consumer
   subject CORRESPONDS: peer
   subject COMBINE: peer
   subject INJECT: aName IN: consumer
       consumer MEMBER: aName ADD: subject
```

The incremental and invasive character of the INR strategy is already visible in the first activity of the interaction strategy, the `MEMBER:PUT:` method. This method is part of LMOP and reifies the composition of one AST node (called the part) as a member of an existing AST node (called the aggregate). The method is specialized by the INR strategy and triggers the INR protocol. The INR strategy is thus executed each time two AST nodes are composed. For example, this is in contrast to the SSQ strategy (in Section 6.3.1) which only extends the functionality of LMOP and which is explicitly triggered by the linglets themselves.

Whenever two AST nodes are composed, potential new locations are available to integrate the nonlocals of the composed AST nodes. The interaction strategy checks in the `INTEGRABLE:` method whether the nonlocals of the composed AST nodes can be integrated. If integration is possible, it is performed by the `INTEGRATE:` method. If integration is not possible, then all the nonlocals from the part are moved to the aggregate by the `RELOCATETO: FROM:` method.

The incremental step, which moves around the nonlocals is performed in the `RELOCATETO: aggregate FROM: part` method. The receiver of such a message is the subject role. The `aggregate` argument is the master role which is played by the successive AST nodes to which the nonlocal is moved. The `part` argument is the node containing the nonlocals.

The `INTEGRABLE:` method determines whether a subject can be integrated in a particular master of the produced program.

The integration of nonlocals defined by the INR strategy is based on the composition interaction strategies of subject-oriented programming (see Section 3.6). Subject-oriented composition establishes correspondences between program elements such as classes and methods, and derives elements of the composed program by combining the corresponding elements. Correspondence is a fairly simple concept: If elements of two subjects describe complementary functionality that should be merged in the combined program, they are said to correspond. For elements that correspond to each other, an element of the combined program is derived from a combination of the corresponding elements. The integration of two program elements is thus a triplet consisting of composition, correspondence and combination rules. The combination of complementary functionality performed by composing their AST representations, nicely captures and defines the integration of multiple results produced by complex translational semantics.

INR adopts the integration triplet in the form of messages exchanged during the protocol. The composition is defined in the `INTEGRATE:` method, the corre-

spondence is defined in the `CORRESPONDS:` method and the combination of two corresponding AST nodes is defined in `COMBINE:` method. The receiver of the integrate message is the subject role, its argument is the consumer role which is played by the selected master. The correspondence is used by the integration to check if nodes in the AST need to be combined with the nonlocal. In the integration of the code fragment, we first check whether the nonlocal already corresponds to an existing part of the consumer by calling the `CORRESPONDs:` method.

If a correspondence is found, the two nodes are combined by calling the `COMBINE:` method. The receiver of these messages is the subject role and the argument is the peer role and is played by the existing part of the consumer. If no correspondence is found, the fragment is added to the consumer using the `INJECT:AS:` method. This method uses the `MEMBER:ADD:` method of LMOP to add the subject to the consumer.

The methods `CORRESPONDS:` and `COMBINE:` are abstract in the INR strategy. They need to be specialized for a particular linglet.

Extension: Context-dependent Integration

We present a small extension of the INR strategy which enables the integration of nonlocals depending on their context (see Section 4.3 and Section 5.9.2).

Protocol - Extension: Context-dependent Integration:

```
subject RELOCATETO: aggregate FROM: part
  subject CONTEXT_CHANGED: master
```

The method `CONTEXT_CHANGED:` is called whenever a nonlocal is moved from one node to another. As such, upon each move, the context of nonlocals can be used to compute their correct integration.

Language Independence

For the combination of program fragments, standard combination strategies such as join and replace [OKK+96] have been developed. The join strategy glues two program fragments together and the replace strategy performs a replace. Glueing can be done in a number of ways such as concatenating, summing, etc. Again, concatenation can be performed by concatenating one structure before or after the other, etc. The point is that there are many rules, which depend on the semantics of the program fragments. The INR offers a language independent implementation. The logic to integrate, correspond, combine or even relocate evidently involves specific semantics of the language or semantics of a particular code fragment. This language-specific logic has to be provided by the actual linglets in the language specification or by the actual code fragment themselves.

Each nonlocal and/or its linglet is thus extended with additional behavior to complete the interaction strategy or to further specialize the interaction strategy. This is done by specializing or adding variations of the following methods:

`CORRESPONDS:, RELOCATETO:FROM:, INTEGRABLE:` stating the specific conditions of where nonlocals should be integrated

`COMBINE:, INTEGRATE:` stating how nonlocals should be added to the rest of the produced program

`CONTEXT_CHANGED` so that nonlocals can take the context into account and adjust their integration.

The specialization of the INR strategy for specific linglets results in an interaction strategy that can take into account the specific semantics of the language which is being implemented by the linglets. As such, more specific and expressive methods and functionality can be offered for handling the produced nonlocal results. The extension possibilities are discussed in more detail in the following sections.

Illustration

Before we dive into the implementation of INR, let us first illustrate how the interaction strategy is used in practice by specifying the semantics of the nonlocal `Table` declaration produced by the `Relation` linglet (Section 5.9.2 on page 207).

Recall the example of the integration of the nonlocal `Table` nodes. It is not possible to judge if a table is discarded solely on the source relation predicate, but the context of the whole expression must be taken into account. In Section 5.9.2 on page 207 we argue that integration logic is more straightforwardly created and more robust when the integration of the nonlocal `Table` nodes is based on the target language. The solution in that section is to specify the integration semantics locally to the linglet that produces the `Table` nodes, by specializing the `context_changed` method.

The method `context_changed` is thus actually part of the INR strategy. For the integration of the `Table` nodes, two other methods need to be specialized: the `corresponds:`, and the `integrable:` methods. The mechanism to relocate, and initiate the integration attempts of the nonlocal `Table` nodes is provided and controlled by (the default implementation of) the INR strategy.

Relation predicates in T2SQL (see Section 5.1) are deeply embedded in the nested expressions specifying the condition to which a set of tuples must adhere to. Hence, in an AST of T2SQL, the `Relation` nodes are deeply nested within the subtree of the source AST that represents the condition of a set. The condition defining a set is transformed into the condition of an equivalent SQL statement. So the translational semantics of the `Relation` linglet is also deeply nested in the

condition of an SQL statement. This is shown in Figure 6.4 which depicts example Query 7 of Table 5.1 and its target query in SQL. The figure also illustrates the integration process of nonlocal `Table` nodes, which we are now about to explain.

Recall from Section 5.9 that the translational semantics of the `Relation` linglet consists of two AST nodes: a `True` node and a `Table` node which is attached to it as a nonlocal. As the `True` node is recursively composed with nodes that ultimately define the condition of an equivalent SQL query, the nonlocal `Table` node is moved upwards in the AST of that SQL query. The successive moves of a nonlocal `Table` in search of a suitable integration location is depicted by the numbered grey arrows in Figure 6.4.

Recall from Section 5.9.2 that the integration of nonlocal `Table` nodes depends on the context nodes `NOT` and `AND` of its producing `Relation` node. Upon each move, the `context_changed` method is called by the INR strategy. When a `NOT` or an `AND` node is encountered, the integration policy of the `Table` node is adjusted. This policy is modelled after a state machine containing two states: `discard` and `required`. These two states are respectively stored as parts `discard` and `required` into the nonlocal. A `NOT` negates the discard and required flag of the nonlocal. An `AND` renders the nonlocal required so its variable is to true.

The INR strategy calls the `integrable:` method in order to determine whether the nonlocal can be integrated in a given master. Whether or not a nonlocal can be integrated depends not only on the fact whether the grammar of the language allows it as it is defined in the INR strategy, but also on the two states defining the integration policy. Therefore, the method `integrable:` is overridden with the following rules stating when a `Table` nonlocal is integrable:

- A nonlocal `Table` cannot be integrated if the grammar of the language disallows this (line 1).
  For example, a `Table` node can only be integrated in the `FROM` clause of a `SELECT` statement i.e. as a `source` part of a `Select` node (cfr. definition of the `Select` linglet in Section 5.6.4). More details about how the grammar is enforced in the INR strategy is given in the next subsection.

- A nonlocal `Table` integration fails if it is in contradictory state, i.e. if it is both required and discardable, or if it is both not required and not discardable (line 2).
  This is, for example, in case in queries 6, 8 and 10 listed in Section 5.9.2.

- A nonlocal `Table` integration drops or ignores the relation when it can be discarded and not required (line 5). Examples of such queries are queries 1, 2, 5 and 9 of Section 5.9.2. Note that the transformation process fails in queries 1,2 and 9 because of at the end of the transformation the equivalent SQL `SELECT` statements contain empty `FROM` clauses.

- A nonlocal `Table` integration fails if the nonlocal is required and there is already an existing table which clashes (line 4) with the nonlocal (line

6). This is the case for example in queries 3 and 4, where the second nonlocal `Table` node `manager w` clashes with the already existing `Table` node `works_on w`.

```
Relation
   nonlocaltable: nonlocal {
      nonlocal discard: false.
      nonlocal required: true.
      nonlocal context_changed: node {
         node linglet type = 'NOT' ifTrue: [
             ast discard: ast discard not
             ast required: ast required not
          ] ifFalse: [
             node linglet type = 'AND' ifTrue: [
                ast required: true
             ].
         ]
      }
      corresponds: master {
         (master linglet type == ast linglet type) and: [
             master alias asString == ast alias asString
         ]
      }
      combinable: master {
         ast required == master required not
      }
      integrable: master {
(1)     label := ast findmember: master.
         (label == nil) ifFalse: [
                "according the grammar a suitable
                 location to integrate the nonlocal table has been found"
(2)            (ast discard == ast required] )
                ifTrue: [ Exception new raiseSignal:
                                  'does not compute' ]
                ifFalse: [
(4)                 clash := (ast member: label) exists:
                          [ :el | (ast corresponds: el) and:
                                  [ ast combinable: el ] ].

(5)                 (discard) ifTrue: [
                         Transcript show: 'Warning: table dropped'.
                         false.
                     ] ifFalse: [  "nonlocal is required"
(6)                      (clash) ifTrue: [ Exception new raiseSignal:
                           'duplicate table,
                            cannot integrate required table' ]
```

```
(7)                                ifFalse: [ true ]
                        ]
          ] ifTrue: [ "label is nil"
                 false.
          ]
      }
   }
```

Implementation

The INR strategy is implemented in the INR linglet (shown below). We recognize the five main methods of the protocol: `relocateto:from, integrate:, corresponds:, combine:` and `inject:in:`.

The INR strategy not only extends LMOP by reflecting on its structure, but also extends its behavior. The strategy specializes LMOP in order to trigger the execution of the strategy so that nonlocals are resolved. The method `MEMBER:PUT:` is specialized so as to try to integrate the nonlocals each time an AST node is composed with another one. The receiver of this message plays the role of the aggregate. For each nonlocal `nl` attached to the part, an attempt is made to integrate it in the aggregate. If the `INTEGRABLE:` method of the nonlocal evaluates to true, the integration is started by calling is `INTEGRATE:` method. Otherwise, the nonlocal is relocated by its `RELOCATETO:FROM:` method.

The `RELOCATETO:FROM:` method removes a nonlocal from its current parts and attaches it to the aggregate.

The `INTEGRABLE:` method evaluates to true when a member of the master is found where the nonlocal can be added to become a part. The member is searched by the `FINDMEMBER:` method. A composition of two AST nodes is valid when it adheres to the grammar of the language. So in order to check whether the nonlocal can become a part of the master, we search for a parameter of the linglet of the master that is bound to the linglet of the nonlocal. If a parameter is found, the corresponding member may include that nonlocal.

The `INTEGRATE:` method stores the nonlocal in the correct member of the master. The current parts of the members are compared with the nonlocal to determine if the two nodes correspond. Correspondence is determined by the `CORRESPONDS:` method. By default, a node never corresponds with another node. If a correspondence is established, the existing part is combined with the nonlocal. The combination of two nodes is implemented by the `COMBINE:` method. There is no default implementation for the combination of two AST nodes, as the actual combination depends on the semantics of the language.

```
Linglet INR {
    member: name put: part {
      previous member: name put: part.
      part nonlocals do:[ :nl |
          (nl integrable: ast)
              ifTrue: [ nl integrate: ast ]
```

```
            ifFalse:[ nl relocateto: ast from: part ]
      ]
    }
    relocateto: aggregate from: part {
      aggregate nonlocals add: ast.
      part nonlocals remove: ast
    }
    integrable: master {
      member := ast findmember: master.
      member ~~ nil
    }
    integrate: consumer {
      member := ast findmember: consumer.
      combined := (ast member: member) exists: [ :el
          (el corresponds: consumer)
              ifTrue: [ el combine: consumer. true]
              ifFalse: [ false ]
      ].
      combined ifFalse:[ ast inject: member in: consumer ].
    }
    inject: part in:  consumer {
      consumer member: part add: ast
    }
    combine: peer { }
    corresponds: peer { false }

    findmember: master {
     master linglet parameters detect: [ :parameter |
       (master linglet parameter: parameter) exists: [ :linglet |
        linglet type = ast linglet type
       ]
     ]
    }
}
```

This implementation is simplified in a couple of ways for clarity. The full implementation with all its extensions (see Section 6.3.3 on pages 244 and 244) is beyond the scope of this text. The most apparent restriction is the shallow integration. This means that the logic to determine whether or not a nonlocal can be integrated in a master only checks whether the nonlocal can become a part of the master. The parts themselves, and in turn their parts are not considered. More details about the possible extensions of this interaction strategy to tackle these restrictions are given in the next subsection.

Due to the genericity of the INR strategy and due to the complexity of integrating

program fragments, there are a lot of opportunities to extend this protocol. We divide the possible extensions according to the two main algorithms of the INR strategy: relocation of the nonlocals and their integration.

We could not include the implementations of all the variations of the INR strategy as the design space of this interaction strategy is simply too large. Discussion of the variations on the INR strategy shows the potential of LMOP to accommodate them. More precisely, we show that an interaction strategy is not monolithic, but rather a space of variations and options, including language-specific logic.

INR Extensions of Relocation

The relocation algorithm only works in cases where the location of nonlocals can be determined unambiguously, or in cases when there is a simple satisfactory interaction strategy to disambiguate. In general purpose target languages like Java or C++, language constructs like classes, datamembers, methods and statements are typical language constructs that are only allowed in specific contexts. E.g. a class cannot be declared in a method body, a variable and a method cannot be declared directly in a package, a statement cannot be part of an expression, etc. Such language constructs are very common in DSLs as well.

In case of domain-specific languages (DSLs) [vDKV00], automatic relocation is very well suited. DSLs are designed to support domain experts, by offering suitable language constructs and a structured way of using them. As the language constructs may only be used in a certain context [EJ01], disambiguation is not often needed.

If the location of other language constructs such as expressions and statements cannot be unambiguously determined, the nearest possible location is often the desired one. However, in languages like Beta and Scheme relocation solely based on the grammar fails for most types of nonlocals, because most language constructs may be used in every other language construct. For these languages, custom relocation must be used.

INR Extensions of Integration

The protocol of the full INR strategy is much more elaborate than the one discussed here. It divides the tasks in smaller units of responsibility or subconcerns, and can handle more complex cases such as:

Generic Integrations There are many composition triplets. In [OKK+96], Ossher et.al. lists a significant set of possible correspondence rules such as equate and MatchByName, and combination rules such as ByNameMerge, ByNameOverride, Before, After, Replace, UseByNameOverride, UseMerge.

Language Construct-Specific Integration Some of the generic rules that integrate, combine and correspond may need to be further refined for specific language constructs. Consider for example the rule to inject statements at the end of the execution of a Java method. It does not suffice to plainly append such statements. One needs to take into account the language constructs that terminate the execution of a method early, such as return statements and exceptions.

Deep integration Except for the leaf nodes of an AST, the AST nodes are subtrees. Integration of a nonlocal into a subtree of an AST node is called deep integration. Deep integration is not simply a recursive version of a shallow integration (which is discussed here). Deep integration may require the creation of intermediate AST nodes before the actual nonlocal can be integrated. Consider, for example, the integration of a statement initializing a datamember in a Java class. The statement cannot become a part of the Java class, it needs to be put inside a constructor of the target class. Hence, the integration must descend into the AST node representing the Java class and subsequently perform the integration. When the Java class does not have an explicit constructor yet, an explicit default constructor must be created first, prior to the integration of the initialization statement. In Section 7.4.10, deep integration is used in our case study where we validate this dissertation.

Multiple integrations Nonlocals that need to be integrated in several locations in the AST require an interaction strategy supporting multiple integrations. So whenever a location in the AST is found in which the nonlocal is integrated, a copy of the nonlocal is integrated so that the integration in other locations can continue.

6.4 Advanced Experiments: Compile-time MOP

The meta-protocol as described up till now is used to establish the necessary collaborations between distant and even unknown linglets so that linglets can effect their complex translational semantics. The meta-protocol is thus used within a language implementation to reflect upon itself. In this section, we use the metaobject protocol, for defining the LTS system itself. More precisely, the structural reflection of the metaobject protocol is used to construct the meta-languages provided by LTS.

From a language perspective, the kernel of LTS defined in the previous chapter consists of two meta-languages: a language to define the linglets and a language to define the specification of languages. The former is called the linglet language (LL), and the latter is called the language specification language (LSL). We consider them as two separate languages because the semantics of the methods of the linglets defined in LL is different from LSL. Exactly how the semantics differs is detailed in Section 6.1.2. As our objective in this dissertation is to conceive and develop a design technique for the modularization of language implementations along language constructs, it is natural to consider developing the two LTS languages separately using our design technique.

By implementing LL and LSL in LTS, these two languages themselves become customizable through LMOP. This gives us the opportunity to add new language constructs to LL and LSL in order to facilitate expressing linglets and language specifications respectively. For example, we can take advantage of syntactic convenience and static checks in the definitions of languages themselves. Especially the benefit of LTS to keep complex language constructs modularized via interaction strategies is now applicable to itself. Since, such customizations of LTS take effect while compiling a language definition L before it processes any concrete program written in L we in fact obtain a

compile-time meta-object protocol of the language L.

We distinguish between the terms *language-compilation-time* and *language-execution-time*. At language-compilation-time, the language specification of a language together with its linglets are compiled by LTS into a run-time representation consisting of LMOP metaobjects. At language-execution-time, this representation is executed with a given source program yielding a target program.

6.4.1 LTS in LTS

LTS is currently implemented as a run-time system that implements the metaobject protocol. So the LTS system is always running, and the LL and the LSL languages are just programs that are run by this run-time system. The implementation of LTS in LTS is illustrated in Figure 6.5. LTS consist of two languages LL and LSL. These respectively compile the linglet definitions and the language specifications to the run-time system. Both languages are in turn implemented using the LTS system. For the language constructs in LL and LSL, linglets are defined which compile to the run-time system, and interaction strategies are defined to handle the interactions among these linglets.

The languages of LTS are good examples for illustrating the qualities of a language implementation in LTS because: first, both languages share part of their language specification, giving us the opportunity to test the reusability and composability of language constructs. Second, their language constructs have complex translational semantics, giving us the opportunity to test the modularization capabilities of LTS.

Reuse of Linglets

Although both languages serve various purposes they have a fairly large part in common as both languages primarily deal with the single concept of a linglet. The LL language creates them and the LSL language assembles them through combination and specialization. As such, the language constructs in both languages contribute to the definition of a linglet.

The bulk of the shared linglets such as `Method`, `Tempories`, `Statement`, `Variable`, `KeywordMessage`, `UnaryMessage`, `BinaryMessage`, `Expression`, `String` and `Integer` define the Smalltalk language. Besides the linglets that define SmallTalk language constructs, there is a new linglet which defines the `#`-construct and two linglets that introduce the pseudo variables `ast` and `previous`.

Although linglets are shared, their combination differs in the LL and the LSL language. This means that a method declaration in the linglet is not semantically equivalent in both languages: In order to preserve isolation, the messages that are defined in LMOP or in one of its extensions may not be called from within a method defined in a linglet. A method declared in the LL may not access distant linglets (not even its parent), while a method in the LSL may do so. In order to enforce the isolation of linglets we need to work both at language-compile-time and at language-execution-time. At language-compile-time, we distinguish between an external method and an internal method, and at language-execution-time, we can check whether a certain method call is allowed from an internal method.

Complex Translational Semantics

The language constructs used in the languages of LTS have complex translational semantics, which gives us the opportunity to test our ability to design these languages using modularized constructs. In the implementation of LL and LSL languages, we encountered linglets that depend on external information which had to be computed using a complex interactions, we encountered linglets that produce multiple results which must become a part of the results produced by other linglets, and we encountered linglets that are facing a compositionality conflict. As an example for such complex translational semantics we discuss left recursion in grammars. It is not only a mere illustration of a compositionality conflict and multiple inputs, but it is an interesting problem in its own right, one that occurs in other language implementations as well.

The parser used by LTS is a simple left recursive descent parser composed of parser combinators [SAA99]. Left-recursive grammars [App98] cannot be implemented with such parsers, because the left recursion in grammars results in an infinite loop. To clarify this consider the following grammar excerpt:

```
Expression ::= Integer
Expression ::= UnaryMessage

UnaryMessage := Expression ID
```

The above grammar defines a part of the LL and LSL language to express unary messages on integers. This grammar is left recursive because the non-terminal `Expression` occurs as the first symbol of the right hand side of an `Expression` production (i.e. the UnaryMessage). The example sentence we use in this section is:

```
3 inc square dec
```

It denotes the square of the increment of three with one, decremented with one. The AST of this sentence using the above grammar, shown in Figure 6.6, is constructed using a series of nested `UnaryMessage` nodes. This nicely illustrates the left recursion of the above grammar.

In a left recursive descent parser each production is implemented with a separate function, and each non-terminal in the right-hand side of productions is an invocation of the corresponding function. In the descent parser for the given grammar, the function `UnaryMessage` immediately calls the `Expression` function, which in turns immediately calls the `UnaryMessage` function, without consuming any tokens from the input stream. Clearly, the parser gets stuck in an infinite loop.

There are simple techniques to rewrite a left recursive grammar into a non-left recursive grammar. Rewriting the above grammar to eliminate left recursion yields the following grammar:

```
(1)   Expression ::= Integer Expression'
(2)   Expression' ::= UnaryMessage Expression'
(3)   Expression' ::=

(4)   UnaryMessage := ID
```

The `Expression` non-terminal is the starting token of a valid expression, (1) consisting of the most concrete symbol `Integer`, followed by a `Expression'`. This latter non-terminal is the remaining part of expressions: (2) consisting of a `UnaryMessage` followed by a `Expression'` or (3) consisting of no token at all. Due to this rewrite strategy, the `UnaryMessage` is reduced to a dependent clause consisting only of the name of the message (4).

Although these abstract syntaxes are mathematically equivalent, there are fewer available parts of a node. Figure 6.6 illustrates this by depicting the ASTs of our example sentence based on the left recursive grammar (a) and on the non-left recursive grammar (b) . In the left AST, `UnaryMessage` nodes are composed with a subtree representing the expression on which the message has to be executed e.g. `inc` operates on the number `3`, `square` on the result of the `inc` message and `dec` on the result of the `square` message. In the right AST, this composition is lost, as `UnaryMessage` nodes are composed with the next message, which is from a semantic point of view, unrelated.

It is important to maintain the original abstract syntax tree, where we wish to be able to define a linglet using left recursion, unaware of the parse problems that might arise when composing that linglet. It is even more importantly a matter of control. The translational semantics of an operator needs to access both left and right parts in order to compute its semantically equivalent value. An example of such a situation is shown in Section 6.4.2, where LSL is extended with a new linglet called `Descendant`. That linglet is a binary operator just like the `UnaryMessage` linglet above. The `Descendant` linglet produces an expression where both the left part and the right part are enclosed as a subexpression.

In LTS, a linglet can be defined as it would be in a left recursive grammar. So unary messages are defined in the `UnaryMessage` linglet like:

```
Linglet UnaryMessage {
  syntax { left operation }
}
```

It consists of two syntactical parameters `left` and `operation` respectively denoting the expression on which an operation operates.

The composition of the `UnaryMessage` linglet in the language specification of LL and LSL can be changed to remove left recursion while still retaining the same abstract syntax tree as it would have been in a left recursive grammar. Like the rewrite technique suggests, we define an `expression'` non-terminal in the linglet `Expression` below. It takes a `left` and optionally a `right` part.

```
Linglet Expression {
  syntax { left !(right) }
}
```

In the language specification, shown below, the start of an expression is defined by composing the `Expression` linglet with an `Integer` and a `BinBranch`. The `BinBrach` is the remainder of an expression which is again an `Expression` linglet composed with `UnaryMessage` and itself to define a series of unary messages.

The `left` parameter of the `UnaryMessage` linglet is bound to the `Nil` linglet. The `Nil` linglet does not have any syntax. Thus, upon parsing, the left part of an unary message can be left blank. As such, the problem of left recursion during parsing is avoided. Figure 6.7 part (a) depicts the resulting AST of our example sentence.

In order to compensate for the loss of the `left` part, we specialize the `left` accessor and retrieve the proper AST node. The left part of `UnaryMessage` is obtained in three steps. The code executing these steps is enumerated in the language specification and correspond to the numbers used in Figure 6.7 (c). The left part of the `UnaryMessage` node is the left part (3) of the expression (2) containing the expression (1) of which the `UnaryMessage` node is a part. The traversal of the AST to retrieve the missing left part of `UnaryMessage` nodes is implemented using the SSQ strategy, rendering the implementation more robust for future changes. The resulting AST is depicted in Figure 6.7 (b).

```
      Expression
        left: Integer.
        right: BinBranch.

      BinBranch=Expression
        left: UnaryMessage.
        right: BinBranch.

      UnaryMessage
       operation: ID.
       left: Nil.
       left: {
(1)      ((ast ancestor: 'Expression')
(2)            ancestor: 'Expression')
(3)                  left.
        }.
```

6.4.2 Compile-time Strategies

As LL and LSL are not fixed, developers can tweak them. In this section, we present an interesting example.

User-friendly syntax

Interaction strategies provide new functionality which can be used by the language developer to better separate the basic language concerns. The new functionality has to be encoded by semantical methods which are general purpose language abstractions. Often the syntax of a method is not the most convenient way of offering the functionality of the interaction strategy.

If we take a look at contemporary language systems, we see that interaction strategies are typically offered together with convenient syntax. Consider for example the

structure-shy queries of XPath which are provided by XSLT. XPath has a path notation as in URLs for navigating through the hierarchical structure of an XML document, hence a path mimics the hierarchical structure of the document.

In our undertaking to implement a part of the XPath strategy in LTS as the SSQ strategy, we apparently lost its syntactic convenience. By extending the LL and the LSL languages, this syntax can be recovered and provided to the language developer. Expressions using the new syntax are translated to method calls to the SSQ strategy. Consider for example the `Descendant` linglet shown below.

```
Linglet Descendant {
   syntax {  left "\\" right }
   generator { | right left |
      right := ast right asString.
      left := ast left generate.
      (ast body linglet hasType: 'ID')
         ifTrue: [ #Expression{ ( 'left )
                              descendant: 'right }  ]
         ifFalse: [   #Expression{ ( 'left )
                                  descendantif: '''right'' }  ]
   }
}
```

This linglet enables the use of expressions like:

```
A \\ B \\ C
```

instead of:

```
(A descendants: 'B') descendants: 'C'
```

This linglet removes a lot of syntactical clutter. First, instead of using the message name, a path operator can be used. Second, it detects if the right part should be enclosed by single quotes, depending whether the argument of the operator is a simple identifier or not. If so, it is treated as a string, and otherwise as an expression yet to be computed. Lastly, the parenthesis to ensure the correct order of evaluation are automatically set.

To conclude, each interaction strategy can be accompanied by an extension of LL and LSL. Hence, developers cannot only apply additional interaction strategies to modularize their language implementations, but also take advantage of additional convenient syntax to use these interaction strategies in a user-friendly way.

Static Checks

Interaction strategies use structural reflection to compute the information they require. Unfortunately, there is no guarantee that interaction strategies will indeed find what they are looking for. However, in many cases we can provide warnings when the required information is guaranteed not to be available.

Let us revisit the XPath example. Given a grammatical description of a document, we can answer the question if a given path will fail to execute or not. For the retrieval of an ancestor of a particular node type, it can be checked whether the current node can have an ancestor of that particular type at all. These and other checks are actually implemented in an extension of LSL.

6.5 Implementing Special-purpose Concerns with Interaction Strategies

The next three sections revisit the tasks and challenges of special-purpose concerns. Interaction strategies are a special mechanism for implementing a particular task or challenge of a special-purpose concern (see Chapter 4). Each section discusses how LMOP can accommodate the needs of the various kinds of special-purpose concerns.

6.5.1 Special-purpose Concern - Compositionality

SP1 - Localized Interventions In this special-purpose concern, localized compositionality conflicts are resolved by changing the composition of AST nodes. In Section 5.7, we illustrate how this can be achieved using the base level LTS. There, by specializing the getters of the parts of linglets, the composition of ASTs is not destructively changed. In other words, the original composition is still retained and can be accessed at the meta level by introspection. For this purpose, the meta level is not forced to go through getters for accessing the parts of an AST node. This is contrary to setting parts of an AST node which always has to be performed through setters (see Section 6.1.6).

The original composition for an AST node is important as this is the primary composition dictated by the grammar which the interactions of other special-purpose concerns may rely on. So, changing the composition non-destructively resolves composition conflicts and does not invalidate such interactions.

For example, linglets that retrieve or store information of other linglets can rely on the grammar composition of linglets in a language specification regardless of their composition which is imposed by their translational semantics. By implementing interaction strategies which rely on the introspection of parts, interactions are not affected by the non-destructive changes in composition.

SP2 - Globalized Interventions Global compositionality conflicts by definition involve several linglets, possibly even all linglets in a language specification. The meta level is ideally suited for covering such a broad scope of linglets, abstracting over the specific parts of AST nodes. So instead of specializing the getters and setters of the various parts, one can specialize all the parts at once.

Conflicts can be resolved at various times during the execution of the transformation process. One can either intercept the introspection of parts, similarly to the localized interventions, so as to compensate for syntactically and semantically invalid compositions. Another possibility is intercepting the modification of the AST nodes and resolve

conflicts when nodes are composed. As such, erroneous compositions are corrected as they occur. Yet another possibility is intervening when the nodes are produced, and altering them so as to avoid erroneous compositions in the future.

A concrete example of interaction strategies for globalized interventions are monads.

6.5.2 Special-purpose Concern - Multiple Inputs

SP3 - Identification Linglets explicitly state their need for external information by declaring abstract methods (see Section 5.3.4). Interaction strategies can anticipate the need for this information by inspecting the meta level of the linglet for abstract methods or can intercept the introspection of unkown requests and respond accordingly.

A concrete example of an interaction strategy for the identification of multiple inputs is forwarding.

SP4 - Obtention of External Information Obtention of external information is available by accessing a global (see Section 6.1.5). In language specifications, global information must be explicitly retrieved and provided, because linglets are isolated from a concrete language implementation. Global information can also be obtained by an interaction strategy that redirects unknown requests and checks if the requested information is available globally. If so, a global value can implicitly become available locally.

SP5 - Obtention of Information of Another Language Concern We distinguish between two approaches used by interaction strategies depending on the location of traversal logic. Either the traversal logic is contained and executed in the linglet that requests information. This is, for example, the case in queries or traversals. In this case, the requesting linglet introspects the parts of AST nodes and/or introspects the parameters of linglets. Another approach is to distribute the traversal logic. Each linglet is capable of processing an information request and redirecting the request to the proper acquaintances. In this case, a linglet intercepts the attempt to retrieve unkown requests and responds accordingly.

Concrete examples of interaction strategies for the obtention of multiple inputs are queries of template-based approaches or traversals (see Section 6.3.1).

SP6 - Obtention of Distributed Information This kind of information is the result of a distributed effort among linglets. So each linglet is equipped with logic tailored for its semantics in order to contribute to the computation of the required information. The linglets involved in this process do not have to be direct acquaintances but can be more distant. In that case, we rely on the same mechanisms discussed in the previous paragraph.

Concrete examples of interaction strategies for the obtention of distributed information are attribute copy- and propagation rules and symbol tables.

SP7 - Provision of Information Information can be provided either when requested or provided when it is computed. Although both scenarios can be dealt with

at the base level, the functionality of the meta level comes in handy when one wants to abstract over many linglets and/or specific kinds of information. In the case where information is provided when required, one relies on the intercession of unkown requests and known parts. In the case where information is provided as it is computed, one changes parts by intercession. Instead of explicitly referring to the name of the information, at the meta level other properties can be taken into account such as structural information of the AST nodes and of the linglets. So the provision may also rely on introspection of AST nodes and their linglets.

6.5.3 Special-purpose Concern - Multiple Outputs

SP8 - Identification via the Target Language Program Computation of the location of nonlocal in the target language program can be quite complex involving detailed semantics of AST nodes. As soon as one moves from the base level to the meta level, AST nodes representing the target program are treated more abstractly. In other words, the target program is reduced to a set of interconnected AST nodes. Hence, moving to the meta level also complicates matters. However, there is also a gain in raising the abstraction level, namely the visibility of the grammar. At the meta level, the grammar is a datastructure which can be used as a primary mechanism for computing the possible locations of nonlocals. Further, disambiguation has to be done by the base level or through a more fine-grained meta level. So identification interaction strategies at the meta level rely on introspection of linglets and AST nodes.

Concrete examples of interaction strategies for the identification via the target language are the INR interaction strategy (see Section 6.3.3) and dynamic rewrite rules.

SP9 - Identification via the Source Language Program The grammar of the source language is typically of little importance for identifying the correct location of nonlocals. However, a limited form is certainly important, such as the type and the behavior of linglets (e.g. the scope of a table declaration in T2SQL (see Section 5.9.2)), which can be used as a distinguishing characteristic to guide the identification based on the source program.

SP10 - Scheduling Scheduling is tackled in LTS by separating the responsiblities of producing and integrating nonlocals. LTS allows the base level to produce nonlocals for unknown target program locations which might not even exist yet. It is the responsibility of custom interaction strategies to monitor the creation and the assignment of target language fragments in order to find and integrate the nonlocals in the proper places. Such interaction strategies can be defined at the meta level in a language independent fashion.

A concrete example of an interaction strategy for the scheduling of nonlocals is the INR strategy (see Section 6.3.3).

SP11 - Integration Using a Three-party Contract Integration of a nonlocal in another AST node is a three party contract in LTS involving the nonlocal itself, the AST node in which to integrate and an external actor specifying the integration

contract. The integration contract can be simplified and made more generic by using the meta level to inspect AST nodes and to analyze their linglets. This information can be fed into the negotiation of the contract. Decisions depending on a potential conflict among the existing parts can be specified at the meta level regardless of the specific AST node involved, by reflecting on the composition of linglets which define the grammar of the target language.

A concrete example of an interaction strategy for the integration using a three-party contract is the INR strategy (see Section 6.3.3). Other integration strategies are implicit node creation and composition rules.

SP12 - Context-dependent Integration The integration of context-dependent nonlocals is potentially influenced by any surrounding AST node both in the source and the target programs. The interaction strategy developer is relieved of explicitly enumerating the relevant nodes. Instead the introspective capabilities LMOP can be used to select nodes based on their metalevel properties to play specific roles in interaction strategies.

A concrete example of an interaction strategy for context-dependent integration of nonlocals is the INR strategy (see Section 6.3.3).

6.6 Discussion

LMOP combines the benefits of two worlds: modularized linglets implementing each language construct in isolation, together with an open-ended list of interaction strategies. We do not have to include new features in the kernel of LTS, nor include additional responsibilities into linglets themselves. Therefore the kernel remains a system with simple semantics, and linglets each capture a single concern. With LMOP, isolated linglets can be used and customized in order to realize complex interactions using interaction strategies.

Language implementations are able to structurally and behaviorally reflect over themselves. Both types of reflection are essential for defining interaction strategies:

Structural reflection grants us the ability to inspect and manipulate the relationships among instantiated linglets, inspect their parameters and parts, inspect and manipulate their behavior. The SSQ strategy inspects the relationships of instantiated linglets for retrieving information. The INR strategy changes the composition of instantiated linglets by injecting nonlocal results. Other interaction strategies like symbol tables add additional behavior to linglets for accessing and storing values which are distributed throughout the compilation process.

Behavioral reflection grants us the ability to change how linglets and their instantiations respond to requests, to intercept how they are combined, how they construct their equivalent target program fragments, etc. The INR strategy changes how program fragments that were produced by other linglets, which do not fit according to the grammar, are handled during language-execution-time. More precisely, the interaction strategy intercepts when two program fragments are combined to initiate the integration of nonlocal results. Other interaction strategies like for

example attribute copy- and propagation rules determine whether a linglet can respond to a request. If a linglet cannot respond to a request then the request is propagated to a neighbor.

LMOP exhibits the following interesting properties to design interaction strategies:

Separate the definition of interaction strategies LMOP is an orthogonal extension of the kernel of LTS. The extension introduces an optional layer of functionality. This precisely meets our objective of separating the implementation of interaction strategies. As there is no silver-bullet interaction strategy which can be anticipated by linglets, multiple interaction strategies have to coexist in LTS. Orthogonality also means that linglets can remain oblivious of any interaction strategy. This is certainly the case for plain one-to-one transformations which do not require any interaction strategy. Moreover, as linglets are involved in interactions initiated by other linglets, a default interaction strategy must be present which is used as a fallback mechanism when no interaction strategy is specified. We have managed to do so for the INR strategy by automating certain interaction strategy decisions e.g. the decision whether or not to integrate is based on the grammar of the target language (see Section 6.3.3).

Scope the application of interaction strategies Both fine-grained and coarse-grained scope control are important in LTS. Metaobjects can change the behavior of a single instance of a linglet. This fine-grained scope control is exercised in the INR strategy to change the behavior of multiple results such that they get integrated in the results produced by other linglets. Recall from Section 5.9.2 that this leads to a localized specification of the integration and identification of nonlocal results.

Metaobjects can change the behavior of a common delegatee[5] among linglets. As such, coarse-grained scope control can be exercised to control the behavior ranging of a subset of interacting linglets, to possibly all the linglets of a language. This granularity of scope control is required to ensure consistency and co-operation between all the linglets of a language. Consider for example the SSQ strategy. Basic structure-shy queries such as descendants and ancestors in LTS do not even need to interact with other linglets, as the client linglet itself can traverse other linglets. Although in principle such an interaction strategy can be defined locally in a linglet, this does not necessarily limit the impact of this interaction strategy to a single linglet. Structure-shy queries issued from a client linglet are in general arbitrary expressions where the basic structure-shy queries are the operators. The successive application of those operators requires that their results (other linglets) also support this interaction strategy. Hence, each linglet of a language can potentially be involved in such an interaction.

Build and customize interaction strategies LMOP defines the execution of LTS in terms of behaviors implemented by linglet metaobjects. Object orientation provides us with the necessary modularity, encapsulation, specialization and refinement required to structure and extend the metaobject layer. The resulting

[5]This term is defined in Section 5.3.8

incremental and ease of use obtained from object orientation enables a layered design which is customized to the concepts of a specific interaction strategy. Furthermore, the default behavior of LMOP can be used in case the behavior is found to be sufficient, otherwise LMOP can be partially extended and specialized. As a result, interaction strategies are not monolithic entities but rather sets of different linglets each capturing a particular feature of an interaction strategy. This is demonstrated in the SSQ strategy in Section 6.3.2, which is designed as a set of extensions of a basic version of the SSQ strategy.

Interaction Strategy binding time The time at which an interaction strategy is bound is called the binding time. This time depends on the scope of the adaption. At the system level and at the linglet level, interaction strategies are bound statically at language-compile-time. At the instance level, interaction strategies are bound at language-execution-time allowing us to incorporate language-execution-time information into an interaction strategy, and tweak an interaction strategy per instance. Both binding times are supported by the prototype-based object orientation through static and dynamic modification of linglets and their instances.

In order to reach a stable LMOP, we implemented a wide range of existing interaction strategies and interaction strategy families:

Traversals AST nodes are visited in a certain visiting order, while operations can be applied on them.

Dynamic Rewrite Rules AST 'rewrite' nodes are threaded along during the execution of linglets. When these match with a node, the matched node is rewritten or modified. These rewrite nodes are removed when they no longer apply. As such, the interaction strategy can scope the effects of these rewrite nodes.

Implicit Node Creation During language-execution-time, each node produced by a linglet is considered a potentially partial state of a program fragment in the final target program. Hence, each node may be merged with another node.

Copy- and Propagation Rules Requests for parts or information to which linglets cannot respond are simply redirected to their parent or an answer is synthesized from the redirections to their parts.

Forwarding Forwarding redirects requests of parts or information to which linglets cannot respond to their semantically equivalent produced target language program fragments.

Queries of Template-based Approaches The best kown query language in LDTs is the set of structure-shy paths provided by XPath in XSLT. XPath query functionality is implemented in the SSQ strategy.

Composition Rules They determine whenever two AST nodes need to be combined. An example of such a rule which depend on the grammar is part of the INR strategy.

Monads determine how semantics of linglets are combined without having to code the combination manually each time it is required.

Symbol Tables This interaction strategy threads information along the execution of the translational semantics of the different linglets.

By implementing the kernel of LTS in LTS using the structural reflection of LMOP we created a compile-time meta-object protocol of LTS. This protocol consists of two languages LL and LSL which define the language to define linglets and define the language to define language specifications respectively. By changing these languages, language developers can tweak how their linglets and language specifications are compiled.

Form this discussion we can conclude that language developers can optimize their language development environment to suit their needs both at language-compilation-time and at language-execution-time. Examples of the latter are interaction strategies, examples of the former are syntactic convenience and static checks.

6.7 Conclusion

Due to their ability to isolate language constructs, linglets having complex translational semantics cannot effect their semantics. In order to allow linglets to effect their semantics without breaking their modularization, we have designed a reflective layer in the form of a meta-object protocol called LMOP. The metaobject protocol is an orthogonal extension of the basic system and thus respects the modularization of the language constructs. In other words, linglets can effect their semantics through LMOP, which requires no changes to be made to their implementation.

LMOP can control the behavior to the level of individual program fragments during the execution of a language implementation in LTS. As such, the translational semantics of a language construct has a high cohesion and a low coupling, as its behavior can be carefully controlled throughout the execution of the language implementation. This fine-grained granularity of LMOP is enabled by the prototype-based object-oriented paradigm.

In order to effect the semantics of a linget, a linglet needs to cooperate with other linglets. Cooperation patterns are lifted to the meta level such that a linglet can reason about and change the information and behavior of other linglets. This is possible through metaobject protocols because the structure and the behavior of the reflective layer mimics the concepts of the basic layer. So a co-operation among linglets at the base level can remain a co-operation at the meta level among the linglet metaobjects.

The meta level and the base level are not stratified. The reason for this lies in the fact that a part of the co-operation which deals with the base level is often directly used in the meta level and vice-versa. As such, co-operations can be implemented in a single entity without unnecessary technical complications.

By separating the actual communication partners of a co-operation in metaobject extensions, metaobject extensions are turned into reusable mechanisms capturing a collaboration with implicit roles, which can implement similar co-operations.

Interaction strategies are not monolithic metaobject extensions. They are modeled as sets of linglet metaobjects each capturing a particular feature of the interaction strategy. Interaction strategies can even be extended with specifically designed features for a particular language implementation. This is an essential feature for some interaction strategies, as in the shift from base to meta level, a concrete language setting is lost. By extending the interaction strategies for particular languages, language specific logic can be incorporated into these interaction strategies while keeping the generic part of the interaction strategy separate and thus reusable.

The suitability, applicability and extensibility of LMOP is illustrated by implementing the SSQ interaction strategy to retrieve information external to the linglets, and the INR strategy to resolve multiple outputs. In addition, we have used LMOP to implement the languages to define linglets and language specifications in the system itself, which gives rise to a compile-time metaobject protocol.

In the next chapter, meta-level implementations of interaction strategies are used to facilitate the design of three different domain-specific languages using a pool of shared linglets. The interaction strategies provide us the means to change the composition of linglets without having to re-implement all the interactions between them.

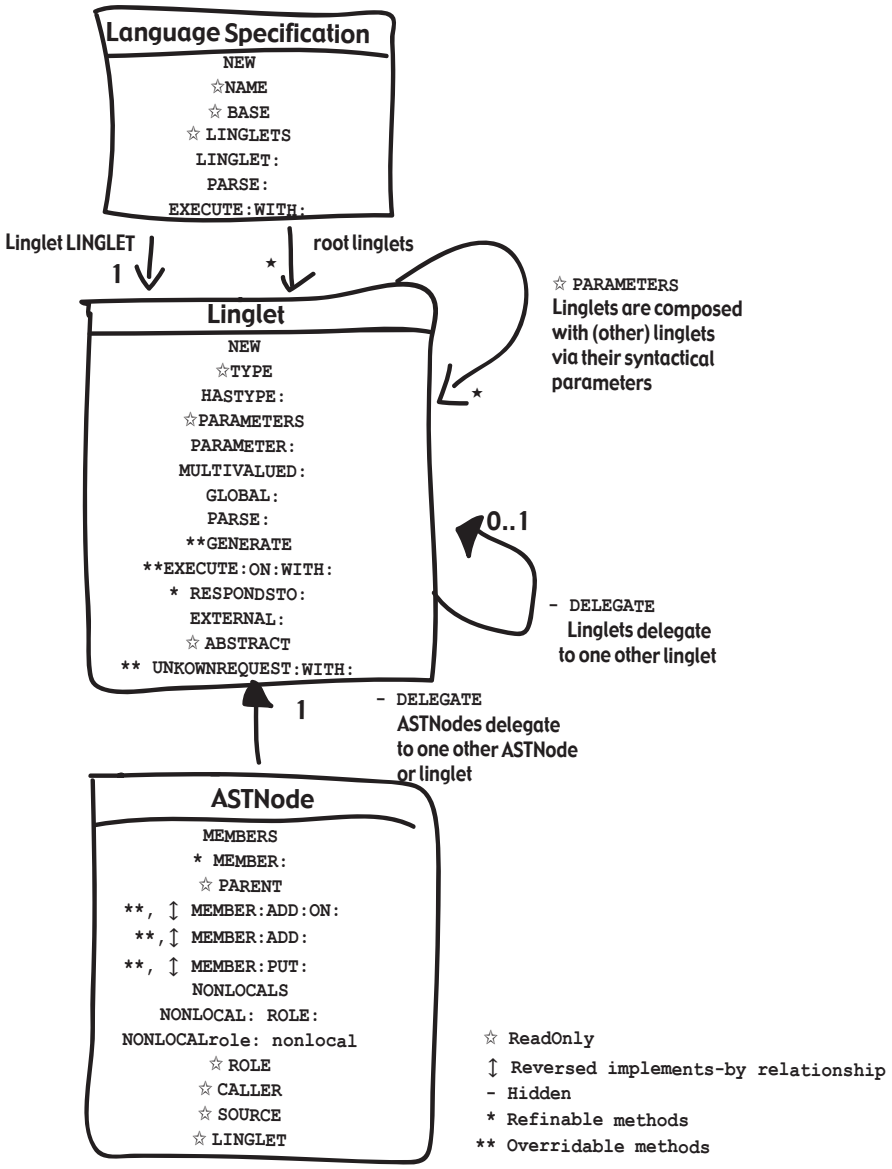

Figure 6.1: Diagram of LMOP.

Sub-protocol - #-Construct

LS BASE PARSE: aProgram LINGLET: aLinglet
aLinglet PARSE: aProgram
Ki PARENT MEMBER: #m PUT: Mi ON: index

Sub-protocol - Nonlocal Results

T NONLOCALS ADD: ti ROLE: role
T NONLOCAL: ti ROLE: role
ti ROLE: role
S NONLOCAL*role*: ti

Sub-protocol - Request Information

EXECUTE: signature WITH: args
MEMBER: signature
MEMBER: signature ADD: arg[1] ON: index
UNKOWNREQUEST: signature WITH: args

Figure 6.2: Subprotocols of LMOP.

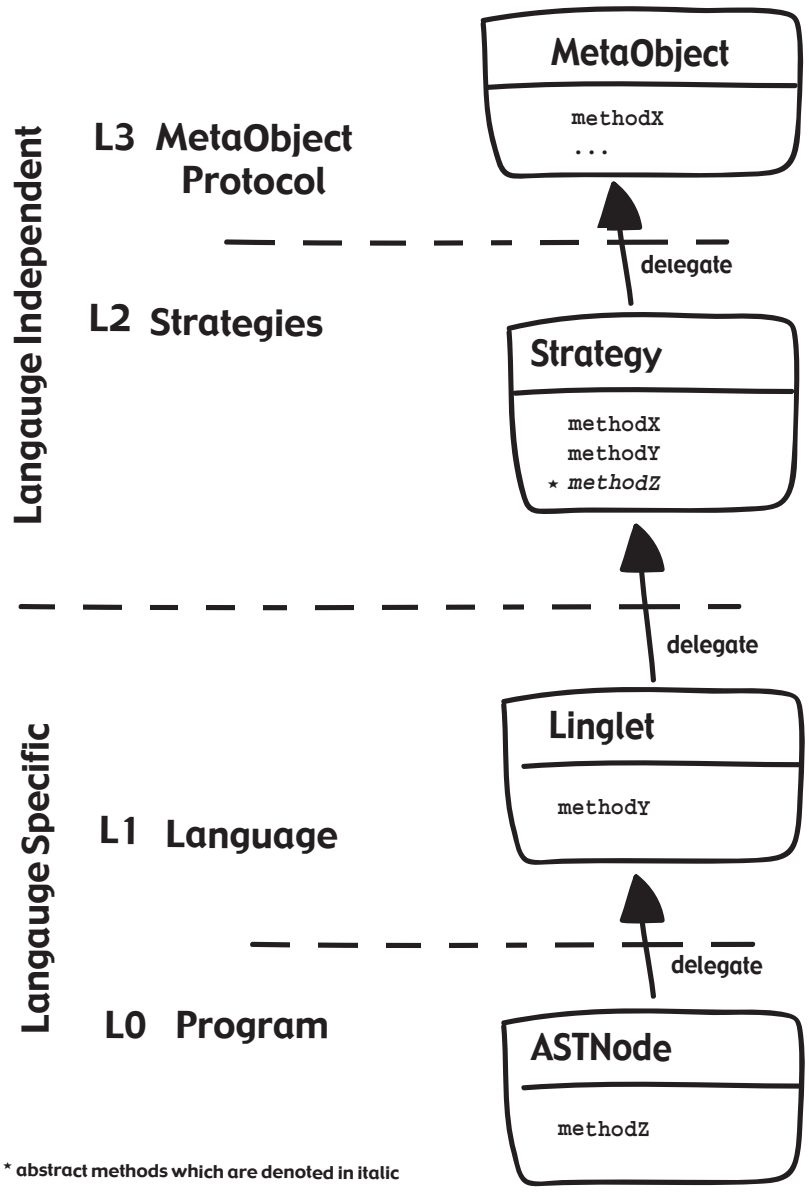

Figure 6.3: Diagram of the deployment of an interaction strategy.

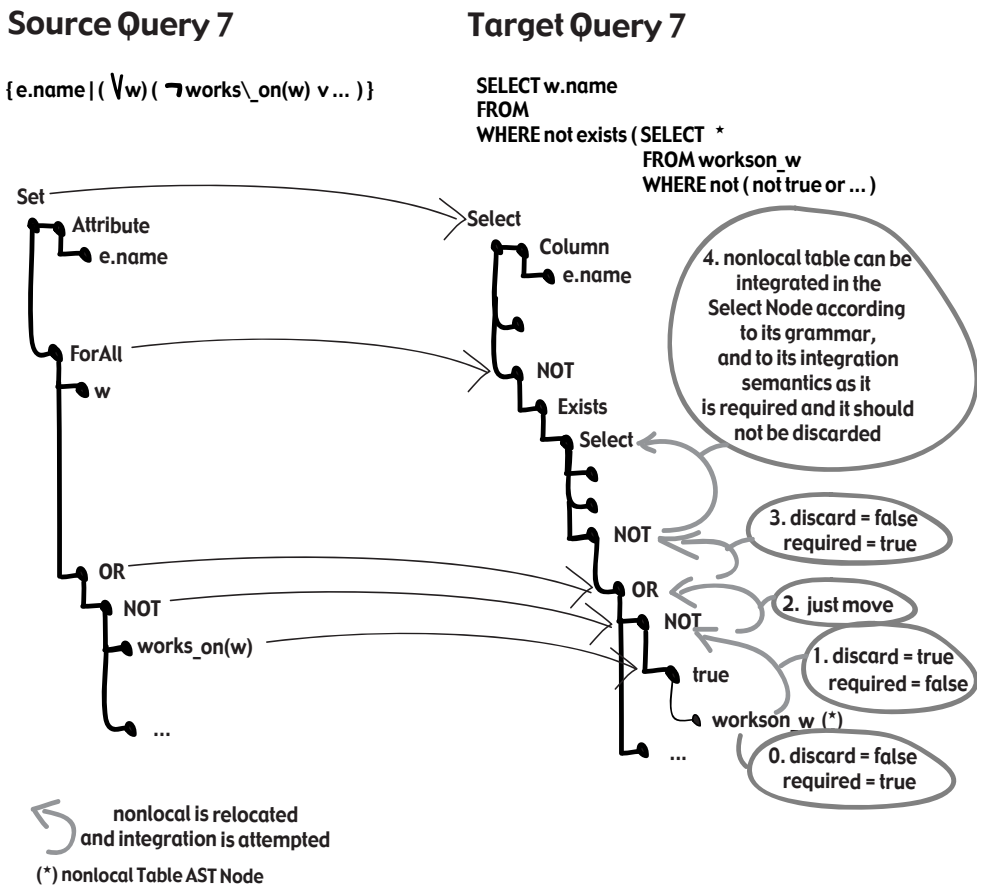

Figure 6.4: Integration of the nonlocal `Table` node `works_on` in example query 7.

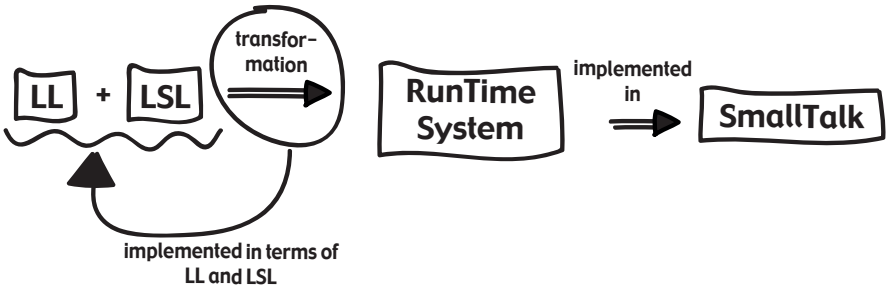

Figure 6.5: Conceptual Diagram of LTS in LTS.

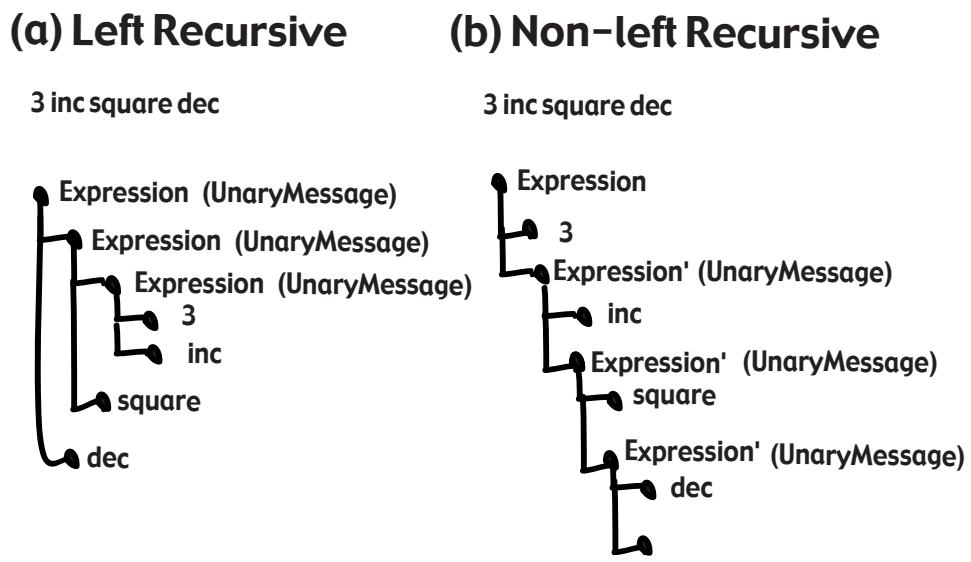

Figure 6.6: Example AST of a left recursively and non-left recursively defined grammar.

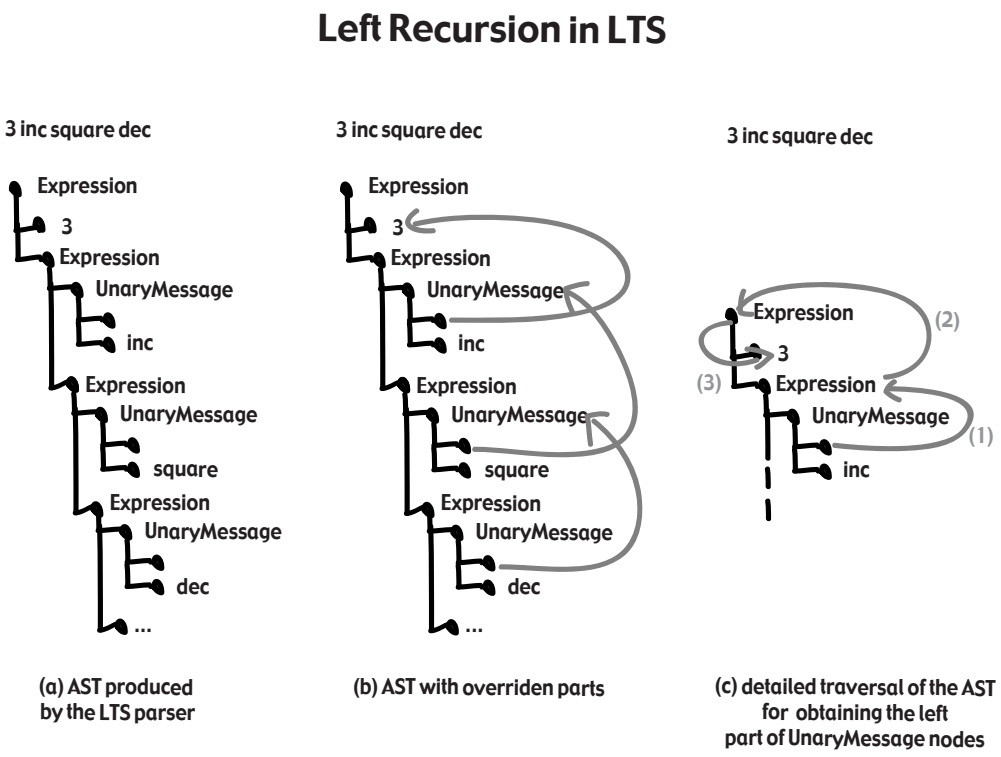

Figure 6.7: Example AST of a non-left recursively defined grammar using left recursively defined linglets in LTS.

Chapter 7

LTS at work: Building a Family of Languages

The goal of this dissertation is to investigate a design technique for modularizing the implementation of languages according to their language constructs. In the previous chapters we explained the principles of this design technique and discussed our implementation called the Linglet Transformation System. In this chapter, we validate the proposed technique.

The case study which we use to validate this dissertation is a family of domain-specific languages for Advanced Transaction Models (ATMS). The languages have been designed by Fabry in his dissertation [Fab05]. We follow an incremental development process and re-implement the languages in the Linglet Transformation System (LTS) that were originally implemented in Java (see Section 5 and Section 6). We grow the compilers of these languages in terms of language constructs and their semantics. Each language is constructed using the same pool of shared basic language concerns and interaction strategies.

We will show that our language implementation design technique (Chapter 5 and Chapter 6) effectively modularizes the implementation of individual language constructs. A thorough evaluation of the gained benefits of such a modularization, which we discussed in the introduction of this dissertation (see Section 1.2), is beyond the scope of this dissertation. This validation is only a first step towards that goal. However, during this validation, we will illustrate the gained benefits of such a modularization:

- *Understandability*: Language constructs are designed and implemented in isolation, focussing on one construct at a time.
- *Evolvability:* Languages co-evolve with their implementations into new languages.
- *Extendibility:* Languages are extended with new language constructs.
- *Reusability:* Language constructs are reused across new languages.
- *Iterative development:* The set of languages are constructed by subsequent iterations.

The language constructs are composed and complemented in a language specification (Chapter 5) to form the various languages. We will show that different compositions and complements of language constructs yield different languages with different overall semantics. The complements are expressed using interaction strategies (Chapter 6). We will show that some interaction strategies need to be customized for improving separation of concerns, others can be reused as such, and yet other interaction strategies are entirely new, designed to tackle the specific separation of concerns problem at hand.

We start with a brief overview of the background of ou case study in Section 7.1. We introduce the concept of advanced transactions and the different models we are about to implement. Section 7.2 details the objectives and the general outline of our experiment. Subsequent sections each present an increment of the stepwise development of Fabry's family of domain-specific transaction languages (DSTLs). Before we conclude this chapter in Section 7.7, Section 7.6 evaluates the experiments.

7.1 Advanced Transaction Models

The concept of transaction management plays a prominent role in many business-oriented systems. Transactions ensure data integrity and free an application programmer from dealing with the complex issue of concurrency management. Transaction management permeates client-server systems that handle a large number of clients which work concurrently on shared data, which usually kept in a database, and that communicates over a wide area network.

A transaction consists of a sequence of program instructions which are considered as an indivisible block. Such blocks, when executed concurrently, may not interfere, and will therefore keep the database in a consistent state.

Transactions have been originally designed to treat small units of work, which merely access a few data items, and take a short time to complete. However, modern client-server applications process large units of work, which renders the basic transaction concept ill-suited. Such transactions take a long time to execute and are more likely to be involved in a deadlock causing many rollbacks, latency time, and the re-execution of other transactions. The reason is that transactions are typically aligned with object-oriented methods, which call yet other methods and consist of several statements to be executed in sequence. Since methods are typically long and call other methods, it is hard to align them with a single transaction.

This is just one example of the mismatch between the properties of a single transaction model and the concurrency management properties requested by modern applications. Other mismatches can be found in [Fab05].

7.1.1 ATMS

A number of *advanced* transaction models (ATMS) have been conceived, so as to address the above mismatches. An impressive number of alternative ATMS can be found in the literature which are brought together in an comprehensive overview by Fabry in [Fab05]. Each ATMS usually treats long-lived transactions in a particular way. Fabry

designed a domain-specific transaction language (DSTL) for a number of ATMS. It is this set of languages that is our case study. A detailed discussion of the different ATMS for which these languages are designed for is given later in this chapter. The list of ATMS we consider in this chapter is:

Classical Transactions

Classical transactions form the most basic transaction model. They declare which methods of applications should behave as a transaction. They are the primary building blocks which are arranged and composed in other ATMS.

Nested Transactions

The best known ATMS is nested transactions [Mos81], which addresses the granularity and scope of rollbacks. Nested transactions arrange a number of transactions onto a call-tree of methods. We distinguish between the case where the hierarchy of transactions is identical to the call hierarchy and the one in which the hierarchy of transactions does not correspond to the call hierarchy.

Sagas

Sagas [GMS87] addresses the issues of long-lived transactions by splitting them into a sequence of sub-transactions called steps. The sequence of sub-transactions should either be executed completely or not at all. This notion will be used to associate a transaction with "big methods" that consist of several statements.

7.1.2 KALA

We have now seen three ATMS, each focusing on a particular shortcoming of basic transactions. Each model can be implemented in terms of the Kernel Aspect Language for ATMS language (KALA) that was proposed by Fabry [Fab05]. In this section we sketch the background and introduce the major concepts of this language.

ATMS can be expressed in a formal model called ACTA. ACTA, however, is solely a formal model and was not conceived with an implementation in mind. KALA is a language that implements the ACTA model allowing developers to separately specify the transactional properties of Java methods. With KALA, different ATMS can thus be specified. Using KALA, a software application programmer declaratively states the transactional properties of a Java method in one block of statements, using the concepts provided by the ACTA formal model. There are also additional constructs present in KALA for using secondary transactions, for naming and grouping transactions and for terminating transactions and groups of transactions.

We do not expect the reader to fully understand KALA. Details are explained when necessary. In the remainder of this section we will briefly introduce the KALA language by means of an example. A more formal overview can be found in Appendix C.

Consider a banking application that transfers money. The money transfer method is conceptually composed of three actions: first performing the actual bank transfer,

second printing a receipt and third updating the logs of the bank. The major issue with the money transfer is the printing out of the receipt: to make a printout, even on a fast printer, takes a few seconds, which effectively turns the transfer method into a long-lived transaction. Therefore, the money transfer transaction is modeled as a saga transaction because it is a long-lived transaction. By splitting the money transfer into a sequence of atomic sub-transactions or steps, the money transfer transaction interferes less with other transactions. As such, by using the Saga ATMS the performance of the banking application can be significantly increased.

Recall that a Saga describes a sequence of sub-transactions called steps. The program, shown below, is an excerpt of a description of a Saga transaction in KALA. The code is the KALA specification for the sub-transaction associated with the `printReceipt(...)` method of the `Cashier` class (line 1). This method is the second step of the Saga money transfer transaction. The money transfer transaction is called the top-level Saga transaction.

```
1 Cashier.printReceipt(Account, Account, int) {
2    alias (Saga <Thread.currentThread()>)
3    alias (CompPrev <""+Saga+"Comp">)
4    groupAdd (self <""+Saga+"Step">)
5    autostart (Cashier.printTransferCancel(Account,Account,int,int)
6     <source, dest, amount, num_receipt> (num_receipt) {
7        name(self <""+Saga+"Comp">)
8        groupAdd(self <""+Saga+"Comp">)
9     })
10  begin {
11    alias (comp <""+Saga+"Comp">)
12    dep(saga ad self, self wd saga, comp bcd self)
13  }
14  commit {
15    alias (comp <""+Saga+"Comp">)
16    dep(CompPrev wcd comp, comp cmd saga, comp bad saga)
17  }
18 }
```

Transactions coincide with the methods of software applications. Hence, the method name of a transaction is the signature of the method implementing the transaction (line 1). In our example Saga step, the method associated with this step is `Cashier.printReceipt(Account, Account, int)`.

The transactional properties (defined in the ACTA formal model) are associated with significant events in the lifecycle of transactions. These are `begin` (line 10), `abort` or `commit` (line 14). Upon begin and commit, in our example Saga step, some transactional properties are stated. In addition, a transaction can also declare some preliminaries i.e. transactional properties which are global to the different events. In our Saga step example at lines 2-9, four KALA statements are defined as preliminaries.

KALA offers naming, so as to set properties on transactions at run-time. Lookup is performed in KALA by using an `alias` statement (line 2), and transactions can be given

a name with the `name` statement (line 7). Transactions can subscribe and unsubscribe to a group so that properties can affect more than one transaction. This is respectively handled by the `groupAdd` (lines 4 and 8) and `groupAlias` statements. Each of these statements take a variable denoting a transaction or a group of transactions and an expression that computes the identifier of the transaction or group of transactions to either lookup or register the transaction or group of transactions. Naming is used in our Saga step example to obtain a reference to its top-level `saga` transaction (line 2) so as to add itself to the group of steps of its Saga (line 4). By adding itself to the group of steps, we ensure that steps are terminated when their Saga ends.

The behavior of multiple, possibly concurrent, transactions is governed by constraints between the events of these transactions. These constraints can be defined by establishing dependencies among transactions. Views relax the conservative visibility defined by the ACID properties of basic transactions. Delegations transfer ownership of the data read and modified by a transaction. For each of these declarations a statement is available: `dep` (line 12), `view` and `del`. Delegations at line 12 are used, in our Saga step example, to trigger the rollback of the Saga when one of its steps rolls back. These set a number of specific dependencies between steps and their Sagas when steps begin to execute (line 12). Furthermore, delegations at line 16 set dependencies that ensure the correct order of execution of the compensation of the step when its Saga has to rollback.

Transactions are not automatically removed from the run-time transaction monitor when they end because other running transactions may have placed dependencies on them. Hence, the programmer must manually declare when these references are no longer needed. For this, KALA offers the `terminate` and `groupTerminate` statements.

Transactions that need to be run separately from the main control flow of software applications are spawned by the `autostart` statement. The execution of these transactions is controlled via dependencies (line 7 - 8). The `autostart` statement consists of four parts. The first part is the method signature to be invoked for compensating the current transaction. The second part lists the actuals which are used to invoke the specified method. The third part lists which of the actuals is a local variable of the method declared as a step transaction (see Section 7.5.2). The final part contains the transactional properties of the compensating transaction. In our Saga step example, the step needs to spawn a secondary transaction (line 5), to be used as a compensating transaction when its Saga rolls back. It achieves this by using an `autostart` statement (lines 5 to 9), which compensates a `printReceipt` simply by performing the inverse `printTransferCancel` method. The secondary transaction adds itself to the group of compensating transactions (line 8), ensuring it is properly terminated when its Saga ends.

7.2 Domain-specific Transaction Languages

The general-purpose nature of KALA comes at a price. Each time a particular ATMS is used in software systems, the whole ATMS has to be reimplemented. Using KALA, the implementation of ATMS is extensive, quite complex, and can lead to a high amount of code duplication [Fab05]. As a result, the use of an ATMS with KALA is difficult

and error-prone. To alleviate this problem, a number of domain-specific transaction languages (DSTLs) have been designed to allow the programmer to use ATMS-related concepts.

In Figure 7.1 we sketch the background of our case study used to validate this dissertation and sketch where LTS comes into the scene. A couple of DSTLs are defined on top of KALA. Each DSTL translates to a KALA specification ((2) in the figure). Together with a software application ((3) in the figure), the generated KALA specification is subsequently fed into the KALA weaver ((4) in the figure) which has been developed by Fabry. The result of this weaving process is pure Java code ((5) in the figure), which contains the software application with transaction code woven into it.

The DSTLs are conceived as a family of languages because there are many opportunities to share and reuse syntax and semantics. This design decision exploits common concepts, minimizing the knowledge required to learn each individual DSTL. Skills acquired by using one DSTL can be transferred to other DSTLs.

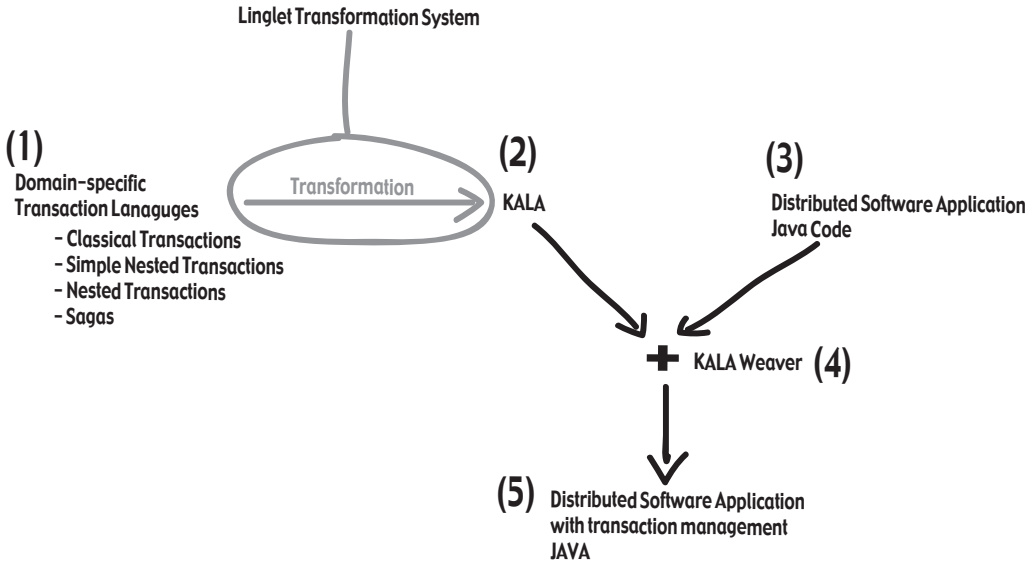

Figure 7.1: Context of the case study.

Fabry designed six languages each time using on average of four language constructs out of eleven language constructs. We re-implemented three of these languages using LTS. The syntax and semantics of the resulting languages are identical to the specification given by Fabry. An early report on the implementation of these languages can be found in [FC05].

The three DSTLs are:

- the ClassicalTx DSTL for the classical transactions ATMS
- the SimpleNestedTx DSTL for a simplified version of the Nested Transactions ATMS

- the Saga DSTL for the Sagas ATMS

7.2.1 Case Study

The implementation of the three DSTLs is the case study presented in the rest of this chapter (see Figure 7.1). It is an interesting case because the three DSTLs share a lot of language constructs and semantics. The case is exceptionally fit to illustrate the benefits of LTS to implement linglets. Each linglet describes the syntax and the translational semantics of the language construct it defines. The compilers of the DSTLs, designed in LTS, maintain the decomposition of the DSTLs into language constructs. The linglets can be composed and reused in the conception of various languages. With this case study we show that in LTS we can design the three DSTL languages using a pool of shared linglets.

The case study is also representative to validate the claims of this dissertation. This is because upon decomposition of the DSTLs and their semantics into their different language constructs, the language constructs exhibit the five phenomena, on which our dissertation is founded and that challenge their modularization[1]:

P0 the translational semantics of their language constructs yield partial KALA fragments

P1 the translational semantics of their language constructs do not compose i.e. the semantics of the parts of a language construct cannot be used to define its semantics;

P2 the translational semantics of their language constructs require multiple inputs, i.e. information which is not present in a language construct but external to the DSTLs or present in other language constructs;

P3 the translational semantics of their language constructs produce multiple results, i.e. multiple KALA fragments which need to be integrated and combined with the KALA fragments produced by other language constructs;

P4 the syntax of some language constructs refer explicitly to the syntax of other language constructs.

Each DSTL is described in a separate section. For each DSTL, we describe the phenomena that we encounter more concretely. We subsequently discuss the ability of LTS to modularize the semantics of the language constructs of the DSTLs facing the above modularization challenges. To this end, we rely on the kernel of the LTS system (see Chapter 5) and on the linglet metaobject protocol (LMOP) of LTS (see Chapter 6).

We formulated for each phenomenon P0 to P4 a requirement R0 to R4 (see Section 4.1.2), in order to ensure the modularization of language constructs where we encounter these phenomena.

[1]The phenomena are discussed in detail in Chapter 4

7.2.2 Incremental Development of Three DSTLs

In the following sections, we discuss the incremental development process used to construct three DSTLs. In each increment we highlight a different feature of LTS which facilitates the *incremental development* of the DSTLs. We relate the LTS features to the requirements R0 to R4 of our formal model, and to the derived design challenges SP1-SP12 (see Section 4.3) to modularize interactions among language constructs.

Initial Language We start with a simple language, called ClassicalTx, describing the classical (i.e. the most basic) transaction model. In this initial language, we show how to construct a linglet defining classical transactions.

Increment 1 The first increment evolves the ClassicalTx language by nesting transactions with an unlimited nesting depth. In the language constructed in this increment, called SimpleNestedTx, the hierarchy of transactions coincides with the call hierarchy in software applications.

Firstly, we show that linglets are able to modularize language constructs despite semantically equivalent partial KALA specifications (P0) and the need for multiple inputs necessary to compute their semantics (P2). Secondly, we show the ability of LTS to overcome compositionality conflicts (P1, SP1) and composition deficit problems. Finally, we show that the modularization of translational semantics (P3) producing multiple results can be maintained due to adequate interaction strategies (SP9-SP12) that establish the necessary interactions to effect such translational semantics.

Increment 2 The second increment creates a new language, called Saga, which focuses not on the nesting depth but on cooperations among sibling transactions in a single level. The idea is to align them with method calls in a method.

In the second increment, we show how multiple results (P3) get integrated by using the source language (SP8). Furthermore, a new interaction strategy is devised which is specific for this case study.

Each DSTL implementation is discussed in a similar fashion. We start by explaining the ATMS which is implemented by that DSTL. We introduce the DSTL by means of an example and discuss its equivalent KALA code in order to demonstrate the complexity of the latter. Subsequently, the translational semantics of the DSTL is formally discussed to indicate the major challenges to modularize its various language constructs. Using this knowledge we present how LTS is able to modularize the language constructs and reuse the language constructs.

7.3 Initial Language: Classical Transactions

As explained in Section 7.1, Classical Transactions is the most basic transaction model. The model defines the primary building blocks which are arranged and composed in other ATMS.

Case

Consider the following excerpt of a method taken from a banking application:

```
class Cashier {
   public void periodicSavings(Account a, Account[] savings,
                               int amount) {
   ...
   }
}
```

The method `Cashier.periodicSavings(...)` is used for periodically transferring the same amount form an account to a set of accounts. The argument `a` is the account from which money is withdrawn, the argument `savings` is an array of accounts on which money deposited, and the argument `amount` is the actual amount of money to be transfered.

Equivalent KALA code

Providing such a method with transactional code in KALA is done as follows. Line 1 declares the transaction, lines 2 and 3 terminate the transaction, on commit or abort.

```
(1)   Cashier.periodicSavings(Account, Account[], int) {
(2)      commit {terminate(self) }
(3)      abort {terminate(self) }
(4)   }
```

DSTL

Using the ClassicalTx DSTL implemented in this section, KALA's complexity for declaring a classical transaction is hidden in a single clause i.e. a transaction declaration. A transaction declaration starts with the keyword `trans` followed by the signature of the method which must be treated as an transaction. The ClassicalTx program, shown below, declares in a single line of code that the method `Cashier.periodicSavings(...)` is transactional.

```
trans Cashier.periodicSavings(Account, Account[], int)
```

7.3.1 DSTL Translational Semantics

Figure 7.2 depicts the translational semantics of the ClassicalTx DSTL. The left side of the figure depicts the DSTL language construct, the right side of the figure depicts its equivalent KALA code fragment. The ClassicalTx language is not very complex as it is described by one language construct: the transaction declaration (`trans`). The transaction declaration is depicted in line (1) in Figure 7.2.

The transaction declaration language construct and its translational semantics is depicted separately by two code templates. The left side being a DSTL code fragment

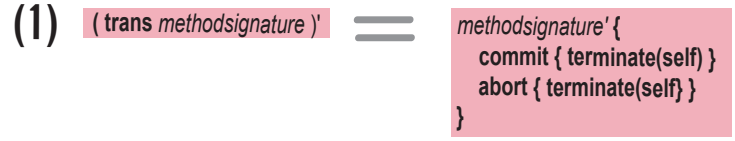

Figure 7.2: Translational semantics of the ClassicalTx DSTL.

of a language construct and the right side being its equivalent in KALA code. The italic names denote template parameters in the source and the target code fragments. The translational semantics of a template parameter is denoted by its corresponding template parameter prime (the name of the parameter postfixed with a quote) e.g. the translational semantics of the template parameter `methodsignature` is denoted by the template parameter `methodsignature'`.

7.3.2 The Tx Language Construct

In LTS, the transaction declaration construct is defined by the `Tx` linglet. The syntax of the `Tx` linglet is straightforward, and the semantics of the `Tx` linglet is a simple one-to-one translation. It does not seem impressive when considering it in isolation. Nevertheless, this initial language allows us to discuss and validate the basic notions when designing a language in LTS.

```
(1)   Linglet Tx {
(2)      syntax {
(3)         "trans" (package ".")* class "." method
(4)                 "(" !(parameter ("," parameter)* ) ")"
(5)      }
(6)      generate { | package class method parameter |
(7)         package := ast package generate.
(8)         class := ast class generate.
(9)         method := ast method generate.
(10)        parameter := ast parameter generate.
(11)        #Tx_Declaration{ 'package.'class.'method('parameter) {
(12)                           commit {terminate(self) }
(13)                           abort {terminate(self) }
(14)                        }
(15)       }
(16)     }
(17)  }
```

The `Tx` linglet contains two methods called `syntax` and `generate`. The first method defines the syntax, the second defines the translational semantics of the transaction declaration language construct.

Syntax

A transaction declaration in the ClassicalTx DSTL takes the form of a `trans` keyword, followed by the method signature of the transaction. The method signature is defined by a series of package names separated by a dot, a class name and a method name followed by a series of parameters enclosed in round brackets. All identifiers within the syntax definition of a linglet are syntactical parameters. As a consequence, the identifiers `package`, `class`, `method` and `parameter` represent the syntactical parameters: package name, class name, method name and parameters. Later, when the language is defined, the syntactical parameters are bound to the `ID` linglet (defined in Section 7.3.3) which defines identifiers.

Translational Semantics

The syntactical description of a linglet implicitly describes its abstract syntax (see Section 5.3.3). The syntactical parameters define the abstract syntax tree (AST). The `Tx` linglet has four parts called `package`, `class`, `method` and `parameter` which can be used to define its translational semantics. This translational semantics is a straightforward one-to-one translation. The transaction declaration is constructed via the `#` construct (lines 11-15) which we introduced in Section 5.3.6. The `#` construct allows a developer to produce a target language program using the concrete syntax of the target language of a translation.

A classical transaction is directly translatable to the language construct `Tx_Declaration` of KALA that defines a transaction (line 11). What remains is a pair of `terminate` statements for removing the current transaction (indicated by the KALA pseudovariable `self`) from the system (lines 12-13). The method signature of the produced transaction declaration consists of a set of metavariables `package`, `class`, `method` and `parameter` which are indicated by the `quote` symbol (line 11). These variables contain the translational semantics of the parts of the `Tx` linglet (lines 7-10). They are initialized in two steps. We first access the accessor of their corresponding parts of the currently executing `Tx` linglet instance or AST node which is denoted by the pseudovariable `ast`. Second, the translational semantics of the parts is computed by invoking the `generate` method.

Observe from the syntax definition that a method signature can consist of multiple parameters and multiple packages. Hence, the `parameter` and `package` variables contain a set of parameters or packages. Like AST nodes, sets of nodes can also be quoted. By quoting we actually refer to the content of the quoted variable, in other words we do not convert the quoted variable to a string. As such, in case of a set of packages or parameters there is no need to know that packages are separated by a dot and that parameters are separated by a comma.

7.3.3 The ID Language Construct

Identifiers in the ClassicalTx language are used for package, class, method and parameter names. Identifiers are defined by the `ID` linglet.

```
(1)   Linglet ID {
(2)      syntax {  (( "a"-"z" [ ast chars add: currentchar ]  |
(3)                "A"-"Z" [ ast chars add: currentchar ]   )
(4)                (( "a"-"z" [ ast chars add: currentchar ]  |
(5)                "A"-"Z" [ ast chars add: currentchar ]     |
(6)                "_"-"_"   [ ast chars add: currentchar ]   |
(7)                "0"-"9" [ ast chars add: currentchar ]   ))*)
(8)             }
(9)      generate { | chars |
(10)         chars := ast chars asString.
(11)         #ID{ 'chars}.
(12)      }
(13)   }
```

The linglet contains two methods called `syntax` and `generate` defining the syntax, and the translational semantics respectively.

Syntax

The syntax of an identifier (line 2) is defined as a series of characters ranging from `a` to `z` (line 4), from `A` to `Z` (line 5), including an underscore (line 6) and ranging form `0` to `9` (line 7). Identifiers can only start with a capital or small letter (lines 2-3). After each character range, a SmallTalk block is specified which adds the current parsed char `currentchar` to the set of characters `chars` of the current AST node `ID`.

Translational Semantics

The semantics of the `ID` linglet is a new identifier of the target language. The new identifier is defined at line 11 using the metavariable `chars`, which contains the characters of the current identifier. The variable `chars` is initialized in two steps. We first access the `chars` part of the current `ID` AST node. Second, this set of characters is converted into a string using the `asString` method.

7.3.4 The Entire ClassicalTx Language

The language specification defining the ClassicalTx DSTL is:

```
name ClassicalTx
base KALA

Tx package: ID.
     class: ID.
```

```
method: ID.
parameter: ID.
```

The first two lines define the name and the target language of the ClassicalTx DSTL. What follows is the introduction and composition of the linglets `Tx` and `ID` linglet in order to define the language.

Linglets are introduced in a language by using their name. Upon introduction, a new linglet is defined which delegates to the definition of the linglet. As such, the syntactical parameters of the new linglet can be bound to other linglets, and methods defined in the linglet can be overridden.

In this language specification, the syntactical parameters `package`, `class`, `method` and `parameter` are bound to the `ID` linglet with the colon construct.

7.4 First Increment: Nested Transactions

In the first increment, we evolve our initial language by nesting transactions with an unlimited nesting depth. This results in a second DSTL called SimpleNestedTx. The discussion of the first increment is structured similarly to the discussion of the previous language.

7.4.1 Nested Transactions ATMS

When methods are being written using a hierarchical calling structure, nested transactions can be used to align with that structure. Rollback of a sub-transaction causes all its sub-transactions to rollback, recursively down the call-tree. In these cases, the scope of a rollback is fine-tuned to the control flow structure of the application. As such, instead of having one big transaction which needs to rollback, smaller transactions can rollback and application code can retry the failed operations. Furthermore, sub-transactions do not commit their work directly but delegate this responsibility to their parent transaction. As such, sub-transactions operate on intermediate data.

7.4.2 Simple Nested Transactions DSTL by Example

In this increment, we start with the most common case where the resulting hierarchy of transactions corresponds to the calling hierarchy. In this case, the nearest parent transactional method is located by going up the call chain.

The new concept apparent in the nested transactions ATMS, over classical transactions, is that a method can be declared as a sub-transaction of the (possibly indirectly) calling transaction. Consider the following code excerpt of a banking application:

```
class Cashier {
   public void periodicSavings(Account a, Account[] savings,
                               int amount) {
      ...
      this.moneyTransfer(a, savings[i],amount)
```

```
      ...
   }

   public void moneyTransfer(Account from, Account to, int amount){
      ...
   }
}
```

Again, the `Cashier.periodicSavings(...)` method periodically transfers an amount of a set of accounts to another account. The argument `a` is the account from which successively money is withdrawn, the argument `savings` is an array of the account on which money is deposited, and the argument `amount` is the amount to be transfered. The `Cashier.periodicSavings(...)` method is implemented by successively calling the method `Cashier.moneyTransfer(...)` to transfer a given amount.

The `Cashier.periodicSavings(...)` method calls for a nested transaction, as the structure of this method is clearly hierarchical and the successive deposits must operate on the intermediate balance of the accounts. If one transfer fails due to a lack of money, all the work performed by the `periodicSavings(...)` method and all the work performed by the `moneyTransfer(...)` must be undone by a rollback.

Equivalent KALA code

The following code taken from Fabry [Fab05] shows how KALA is used to annotate Java code with ATMS code to reflect such nested transactions.

```
(1. T/N)   Cashier.periodicSavings(Account, Account[], int) {
(2. N)        name(self <Thread.currentThread()>)
(3. T)        commit {terminate(self) }
(4. T)        abort {terminate(self) }
(5. T/N)   }

(6. T/E)  Cashier.moneyTransfer(Account, Account, int) {
(7. E/C)     alias(parent <Thread.currentThread()>)
(8. E)        name(self <Thread.currentThread()>)
(9. E)        begin {
(10. E)           dep(self wd parent, parent cd self)
(11. E)           view(self parent) }
(12. T/E)     commit {
(13. E)           del(self parent)
(14. E/C)         name(parent <Thread.currentThread()>)
(15. T)           terminate(self) }
(16. T/E)     abort {
(17. E/C)         name(parent <Thread.currentThread()>)
(18. T)           terminate(self) }
(19. T/E)  }
```

The code at lines 1 to 5 is the KALA equivalent of the root transaction of the DSTL program, and the code at lines 6 to 19 is the KALA equivalent of the sub-transaction.

The code uses the current thread object[2] to identify the calling transactional method. The root transaction will register itself using the current thread object (line 2). The child transactions look up their parent using current thread object as name (line 7). These transactions save the parent identifier locally and register themselves as current thread object (line 8), to enable their sub-transactions to obtain a reference to them in their turn. At the end of the transaction, the identity of the parent is restored using the saved identifier (lines 14 and 17).

At the beginning of a sub-transaction, a couple of dependencies are set so that if the parent aborts, the sub-transaction aborts as well (`wd` dependency line 10) and so that the parent does not commit until the sub-transaction has committed (`cd` dependency line 10). Upon commit, the sub-transaction delegates the responsibility for committing or aborting the data to the parent (line 13). Finally, a sub-transaction has a view on the intermediate results of its parent, which is achieved by setting the view at the beginning of the sub-transaction in line 11. For more details we refer to [Fab05].

DSTL

The complexity of this KALA program is hidden by our second DSL which explicitly supports nested sub-transactions. Sub-transactions are declared by reusing the transaction declaration from the ClassicalTx DSTL form Section 7.3.2, and extending it with the `extends caller` statement. The keyword `caller` is an expression to denote that the parent transactional method can be found in the dynamic call chain.

Root transactions can be stated explicitly in the DSTL as well, and as they do not extend any other transaction, they are therefore specified by omitting the `extends caller` statements. Root transactions are declared using the `trans` clause of the Classical DSTL. Line 1 declares the method `Cashier.periodicSavings(...)` as the root transaction. Lines 2 and 3 declare the method `Cashier.moneyTransfer(...)` as a sub-transaction of the nearest transactional method, which is in this case, according to the control flow structure of the example application code, the `Cashier.periodicSavings(...)` transactional method.

The following code excerpt shows how the above KALA code is hidden in our DSTL.

```
(1)   trans Cashier.periodicSavings(Account, Account[], int)
(2)   trans Cashier.moneyTransfer(Account, Account, int)
(3)      extends caller
```

7.4.3 DSTL Translational Semantics

Let us now have a look at the translational semantics of the SimpleNestedTx DSTL which is shown in Figure 7.3. Again, the left-hand side of the figure depicts source code templates for the two basic concepts of the language, the right-hand side of the figure is the equivalent KALA code for these concepts. The first concept is a root transaction

[2] `Thread.currentThread()` is used because it uniquely identifies the current control flow in Java.

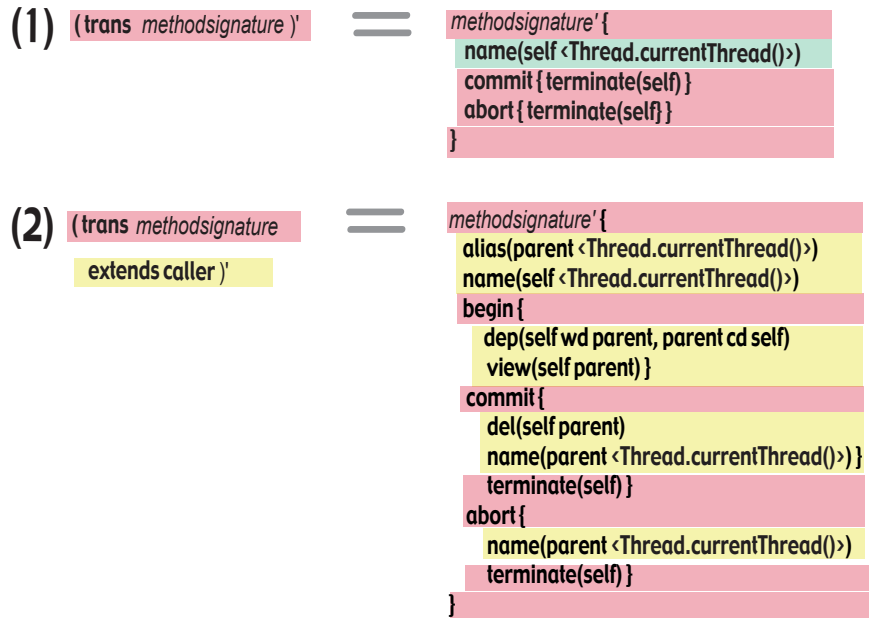

Figure 7.3: Translational semantics of the SimpleNestedTx DSTL.

((1) in the figure), the second is a sub-transaction ((2) in the figure). We now discuss each of the concepts in more detail:

Root Transaction In the KALA equivalent of our SimpleNestedTx program, the root transaction `Cashier.periodicSavings(...)` contains a `name` statement that registers the root transaction so that sub-transactions can refer to their parent. Hence, from a syntactical and conceptual language point of view, the root transaction is a classical transaction. However, the translational semantics of a root transaction of the SimpleNestedTx DSTL and a classical transaction of the ClassicalTx DSTL, shown in Figure 7.2, do not correspond.

Indeed, the `name` statement is not present in the semantics of a classical transaction. In the translational semantics of a root transaction of the SimpleNestedTx DSTL ((1) in Figure 7.3) we marked this additional `name` statement in green. Obviously, we cannot simply reuse the transaction declaration of the ClassicalTx DSTL. The SimpleNestedTx DSTL compiler needs to invasively change the semantics of a classical transaction declaration.

Sub-transaction The translational semantics of a sub-transaction is a complex KALA program ((2) in Figure 7.3). The additional `extends caller` clause, which is marked in yellow, invasively changes the translational semantics of a classical transaction. As such, defining the semantics of a sub-transaction, reusing the semantics of the classical transaction declaration and the semantics of the extends clause requires to invasively compose two KALA transaction specifications. We

recognize this phenomenon as an instance of multiple results (R3) we defined in Section 4.1.5.

7.4.4 Overview of the Language Implementation in LTS

In the SimpleNestedTx DSTL we distinguish between two language constructs, each implemented by their respective linglets: the transaction declaration (`trans` keyword) and extensions of a transaction (`extends caller` keyword). In LTS, the transaction declaration is defined by the `Tx` linglet (see Section 7.3.2) and the extension of a transaction is defined by the `Extends` linglet (see Section 7.4.5).

The `Tx` linglet was already defined in the ClassicalTx DSTL, and therefore we want to reuse it. However, as we explained in the previous section, we require an additional KALA statement that is marked in green (see Figure 7.3). To be able to reuse the `Tx` linglet we create an additional linglet which produces the additional green KALA statement. This linglet is called the `TxRegistration` linglet (see Section 7.4.6). Hence, the linglets of the SimpleNestedTx language comprise the `Tx`, the `Extends` and the `TxRegistration` linglets.

As we explained and illustrated in Section 2.3.1, in general, the more fine-grained transformations are, the more interactions are necessary among these transformations to yield the desired language semantics. The modularization challenges are due to the fact that interactions must be defined separately from transformations in order to maintain their modularization. The same is true for linglets. Due to the fine granularity of linglets, their complexity, and the challenges to modularize them increases. In our increment we encounter, linglets that do not compose syntactically, that do not compose semantically (R1), that consume multiple inputs (R2), that produce partial KALA fragments (R0) and that require to combine and integrate their partial KALA fragments (R3). All of these problems are described formally in Chapter 4.

Due to the modularization of language constructs the interactions among constructs, more precisely the combination and integration of the KALA fragments, are not part of the linglets that define them. These interactions are implemented separately with the INR strategy (see Section 7.4.10) which extends the linglet metaobject protocol such that the linglets are not polluted with combination and integration semantics.

The SimpleNestedTx DSTL will be defined in a language specification (see Section 5.4) that combines linglets and extends LTS for implementing the interactions among its linglets (see Section 7.4.10).

7.4.5 The Extends Language Construct

The first language construct in the SimplesNestedTx DSTL defines sub-transactions. Recall that sub-transactions are declared by reusing the classical transactions of the ClassicalTx DSTL, and extending it with the `extends caller` statement. The extension is defined by the `Extends` linglet, in full accord with the semantics depicted in line 2 of Figure 7.3.

```
(1)   Linglet Extends {
```

```
(2)      syntax {
(3)         "extends caller"
(4)      }
(5)      generate { | methodsignature |
(6)            methodsignature := ast methodsignature generate.
(7)            #Tx_Declaration{ 'methodsignature {
(8)                 alias(parent <Thread.currentThread()>)
(9)                 name(self <Thread.currentThread()>)
(10)                 begin {
(11)                    dep(self wd parent, parent cd self)
(12)                    view(self parent) }
(13)                 abort {
(14)                    name(parent <Thread.currentThread()>) }
(15)                 commit {
(16)                    name(parent <Thread.currentThread()>)
(17)                    del(self parent) } }
(18)           }
(19)     }
(20)     methodsignature { nil }
(21)  }
```

Modularizing semantics: scattering of results

The `generate` method defines the translational semantics. The `Extends` linglet produces a KALA transaction, which we refer to as the *extends transaction*, that contains the transactional properties to turn a transaction into a sub-transaction. We did not chose to produce only the KALA fragments as have they conceptually belong with another. So the idea is apparently to merge two KALA declarations in the end. Rather than to insert individual KALA statements.

We do not expect the reader to understand the semantics of the KALA statements of the extends transactions. Details are given in Section 7.4.2. The transaction fragment is constructed using the `#` construct (see Section 5.3.6) (lines 7-18)[3].

The produced extends transaction is defined in full accordance with the semantics depicted in line 2 of Figure 7.3. The semantics of the `extends caller` clause invasively changes the semantics of the classical transaction declaration. In LTS, we are able to modularize the semantics of this clause such that it is defined independently from the semantics of classical transaction declaration defined by the `Tx` linglet. The `extends` linglet just produces a new transaction fragment containing additional transactional properties that are needed to turn a classical transaction into a sub-transaction. In other words, the `Extends` linglet produces the code marked in yellow.

[3]A KALA transaction is defined by the `Tx_Declaration` linglet of our LTS KALA language implementation (see Appendix C)

Modularizing semantics: partial results

Just like any other KALA transaction, the extends transaction requires a method signature. As the syntax of the `Extends` linglet has no syntactical parameter, the linglet cannot obtain the method signature for the extends transaction. The need for a method signature is stated in the `methodsignature` method defined at line 20.

We have chosen to simply omit the method signature, the `methodsignature` method returns `nil` (line 20), and leave the extends transaction incomplete. This is an example of the R0 requirement (see Section 4.1.6). It is non-essential information, which may be, in full accord with the model requirement (R1a), omitted by linglets. In other words, the `Extends` linglet does not to need concern itself on how to complete the extends transaction.

7.4.6 The TxRegistration Language Construct

As sub-transactions refer to their parent, root transactions need to register themselves with a name. However, as we already showed, solely using the classical transaction to define a root transaction does not work because a classical transaction does not register itself (see Section 7.3.2). The `TxRegistration` linglet produces the additional KALA statement which is marked in green in line 1 of Figure 7.3.

```
(1)   Linglet TxRegistration {
(2)      syntax { body }
(3)      generate { | methodsignature |
(4)         methodsignature := ast methodsignature generate.
(5)         body := ast body generate.
(6)         #Tx_Declaration{ 'methodsignature {
(7)                          name(self <'body>)
(8)                          }
(9)         }
(10)      }
(11)      methodsignature { nil }
(12)   }
```

The first method (line 2) defines the syntax. The syntax consists of one syntactical parameter, called `body` denoting the expression which is used to register a transaction. Hence, the linglet does not define additional syntax concrete syntax[4].

Modularizing semantics: scattered results

The second method (line 3) defines the translational semantics. The `TxRegistration` linglet produces a transaction of its own (line 6) which registers itself (line 7) using the name computed by the `body` part of the `TxRegistration` linglet (line 5).

[4]In context free grammar terminology, we speak about a unit production i.e. of the form `A ::= A`.

The code produced by the `TxRegistration` linglet is the code marked in green (see Figure 7.3). As you can see the produced code should be part of the root transaction. Recall that the root transaction is produced by the `Tx` linglet. So the semantics of the `TxRegistration` linglet actually should inject a KALA statement in the KALA transaction produced by the `Tx` linglet.

Now, in order to ensure the modularization of the `TxRegistration` linglet, the linglet only produces a new transaction declaration, independently of the `Tx` linglet. We refer to the produced transaction as the *registration transaction*. As such, we preserve the modularization of the `Tx` linglet and the `TxRegistration` linglet.

Modularizing semantics: partial results

The `TxRegistration` linglet has one part called `body`, which can be used to define its translational semantics. However, as any KALA transaction, the extends transaction produced by the `TxRegistration` linglet also requires a method signature. Similar to the `Extends` linglet, the `TxRegistration` linglet requires an additional part i.e. the method signature of the registration transaction (line 6, `methodsignature`). This is stated by the third method (line 11).

We can simply omit the method signature and leave the new transaction fragment incomplete: the method `methodsignature` returns `nil`. This an example of the R0 requirement (see Section 4.1.6). It is non-essential information, which may be, in full accord with the model requirement (R1a), omitted by the linglet.

7.4.7 The Entire SimpleNestedTx Language

The language specification defines the SimpleNestedTx DSTL, by introducing and composing the linglets `Tx`, `Extends`, `TxRegistration` as follows:

```
(1)   name SimpleNestedTx
(2)   base KALA

(3)   Root body: NestedTx. RootTx.

(4)   RootTx=TxRegistration
(5)     body: Tx.
(6)     body: { #Expression{ Thread.CurrentThread() } }
(7)     generate : {  | classicaltx registrationtx |
(8)                   registrationtx := previous generate.
(9)                   classicaltx := (previous body generate).
(10)                  classicaltx nonlocals add: registrationtx.
(11)                  classicaltx  }.

(12)   NestedTx=Concat left: Tx. right: Extends.

(13)   Tx package: ID.
(14)        class: ID.
```

```
(15)        method: ID.
(16)        parameter: ID.
```

The first two lines declare the name of the SimpleNestedTx DSTL and the target language. The first linglet introduced in the language is the starting linglet or root linglet for the language. It serves as the starting point to parse programs and to initiate the transformation process. The first linglet in the SimpleNestedTx DSLT is the `Root` linglet. It is a language independent linglet which basically allows programs to consist of a series of sentences defined by the linglets which are bound to it (line 3). The SimpleNestedTx DSTL allows two kinds of phrases to be built: a root transaction without an extension (`RootTx` linglet) and a transaction with an extension (`NestedTx` linglet).

The definitions of the `RootTx` and `NestedTx` linglets using the `Tx`, `TxRegistration` and `Extends` linglets is non-trivial. Due to the isolation of these linglets three problems need to be solved in order to compose linglets into a valid language. The first problem is a compositionality conflict. The second problem is a composition deficit. The third problem is the invasive composition of the produced KALA specifications. In the following three sections, we demonstrate the ability of LTS to overcome these problems.

7.4.8 Root Transactions: Compositionality Conflict

Recall from Section 7.4.3 that in line 1 of Figure 7.3, the translational semantics of root transactions is the combination of the semantics of a classical transaction (red code) with an additional `name` statement. The former is defined by the `Tx` linglet and the latter by the `TxRegistration` linglet. Therefore root transactions are defined by the `RootTx` linglet as an alias, using the `=` operator, for the composition of the `TxRegistration` and `Tx` linglets.

However, due to the isolation of the `TxRegistration` and `Tx` linglet, a compositionality conflict arises when these linglets are composed. This is caused because the translational semantics of the `TxRegistration` linglet cannot be composed with the semantics of the `Tx` linglet. The semantics to resolve the composition conflict is written in the language specification (lines 7-11) in order to preserve the modularity of the linglets involved. The `TxRegistration` and the `Tx` linglets are composed in four steps:

1. The `TxRegistration` linglet is introduced in the language specification in line 4. Recall from Section 5.4.2, that upon introduction of a definition of a linglet, a new linglet is defined which delegates to the defined linglet. The newly created linglet is aliased as the `RootTx` linglet, and is specialized with behavior to resolve the compositionality conflict.

2. The `body` parameter of the `TxRegistration` linglet is bound to the `Tx` linglet (line 5). Syntactically this makes sense as a root transaction (`RootTx` linglet) is a plain transaction. So whenever the DSL programmer declares a root transaction, he/she will implicitly also use the `TxRegistration` linglet. As such, each root transaction will register itself.

3. From a semantical perspective the `body` parameter of the `TxRegistration` linglet must actually be bound to the expression yielding the name to register the transaction. However, as the `body` parameter of the `TxRegistration` linglet is bound to the `Tx` linglet a compositionality conflict occurs: the semantics of the `body` part cannot be substituted in the semantics of the `TxRegistration` linglet because the translational semantics of `Tx` is not an `expression`, but a complete transaction. As we explain in Section 5.7, in order to solve this problem in LTS, we just need to override the getter of the `body` part (line 6) and provide an `expression` that yields the name to register the transaction. Hence, the compositionality conflict between the `TxRegistration` and the `Tx` linglet is resolved, without invasively changing the involved linglets or in other words violating the modularity of the involved linglets.

4. The translational semantics of the `TxRegistration` linglet produces and returns a new registration transaction (code line 6 of the definition of the `TxRegistration` linglet in Section 7.4.6). However, as we stated earlier, we require the combination of the transactions produced by the `Tx` and `TxRegistration` linglet. For this, the `generate` method of the `TxRegistration` linglet (line 7-11) is overridden. In that method we retrieve the two transactions and combine them.

   The registration transaction, which is stored in the variable `registrationtx`, is the result of the original `generate` method which is obtained by the expression `previous generate`. The classical transaction produced by the `Tx` linglet, which is stored in the variable `classicaltx`, is obtained by invoking the original getter for the `body` method and subsequently retrieving its translational semantics by invoking the `generate` method.

   In LTS, we do not have to manually combine the registration transaction with the classical transaction, as that would introduce too much coupling between the `TxRegistration` linglet and the `Tx` linglet in the language specification. Instead the `registrationtx` node is simply attached as a nonlocal result (cfr Section 5.9) to the `classicaltx` node in line 10. Hence, we can ignore the need to combine the two nodes. A separately defined interaction strategy, called the INR strategy, takes care of properly combining them (see next section).

The composition of the `TxRegistration` and the `Tx` linglet in the SimpleNestedTx DSTL shows the versatility of LTS to overcome compositionality conflicts.

7.4.9 Nested Transactions: Composition Deficit

Recall from Section 7.4.3 that in line 1 of Figure 7.3, the semantics of nested transactions is the combination of the semantics of a classical transaction (red code) with the semantics of the `extends caller` clause. The former is defined by the `Tx` linglet and the latter by the `Extends` linglet. Therefore nested transactions are defined by the

`NestedTx` linglet as an alias, using the `=` operator, for the composition of the `Tx` and `Extends` linglets.

Due to the isolation of linglets, they sometimes have too few syntactical parameters to compose them with other linglets. We call this problem a *composition deficit*. The composition deficit problem encountered in the SimpleNestedTx DSTL occurs between the `Tx` and the `Extends` linglet (see also Section 7.4.5).

The `Tx` linglet has several parameters denoting the signature of the transactional method and the `Extends` has no parameters. There are thus no parameters left to compose the `Tx` and the `Extends` linglet. In order to solve this problem in LTS, while still reusing the linglets and maintaining their modularity, we use auxiliary linglets. Such linglets are language independent and are especially designed to overcome composition problems.

The composition of the `Tx` and the `Extends` linglet is realized by the auxiliary `Concat` linglet, which is shown below. The `Concat` linglet has two parameters: `left` and `right`. Its semantics is to compose the translational semantics of both parts. Because we modeled the `Contact` linglet as a language independent linglet, the `Contact` linglet does not know how to compose the translational semantics of the `Tx` and the `Extends` linglet. So the default composition (line 8) hooks the semantics of the `right` part to the `left` part as a nonlocal (line 9), and returns the latter (line 10). By hooking AST nodes as nonlocals to another AST node, the actual composition of the two nodes can be temporarily ignored and later in the transformation process the nonlocals can be combined. A separately defined interaction strategy, called the INR strategy, takes care of properly combining them (see next section).

```
(1)   Linglet Concat {
(2)      syntax { left right }
(3)      generate { | left right |
(4)         left := ast left generate.
(5)         right := ast right generate.
(6)         compose: left with: right
(7)      }
(8)      compose:left with:right {
(9)         left nonlocals add: right.
(10)        left
(11)     }
(12)  }
```

7.4.10 Integration of Transaction Fragments: INR strategy

In this section, we show that the modularization of complex translational semantics (see Definition 4.4) producing multiple results (R3) can be maintained due to adequate interaction strategies (SP9-SP12) that establish the necessary interactions to effect the complex translational semantics.

Throughout the discussion of the increment, we frequently relied on the INR (see Section 6.3.3) interaction strategy to compose and integrate the different transaction

fragments. In our discussion of the semantics of the SimpleNestedTx DSTL in Section 7.4.3 we illustrated in Figure 7.3, that the `extends caller` clause requires invasive changes to be made to the classical transaction and that an additional statement needs to be injected in the semantics of a classical transaction. In this section, we explain how we can compose and integrate the KALA fragments by deploying the INR strategy. For a full and detailed discussion about this interaction strategy, we refer the reader to Section 6.3.3.

Maintaining modularization of the linglets

The semantics of the different linglets `Tx`, `TxRegistration` and `Extends` is complex but modularized nevertheless. Each linglet produces fragments of a transaction specification (R3) (see Figure 7.3) which needs to be integrated and combined into a single transaction specification (SP9).

Neither the linglets nor the language specification of the SimpleNestedTx DSTL contains statements regarding the combination and integration of the produced KALA transactions. It is by extending the linglet metaobject protocol (LMOP) of LTS with the INR strategy (see Section 6.3.3), that we can integrate and combine different transaction specifications without having to pollute or change the semantics of the linglets which produced them.

INR strategy with deep integration

An interaction strategy is a separately defined meta-level mechanism to implement interactions between linglets. The INR strategy is a mechanism to integrate and combine AST nodes. We have defined this interaction strategy in Section 6.3.3 in the `INR` linglet.

The INR strategy pairwise integrates the produced KALA transactions until a single KALA transaction is produced. In order to integrate the produced KALA transactions, the INR strategy must support deep integration (see Section 6.3.3) which means that the parts of the AST nodes representing two KALA transactions are recursively combined. We described an extension of the INR to support deep integration in the linglet called `Deep` in Section 6.3.3.

The INR strategy changes LMOP so that upon the composition of two AST nodes which we respectively call `master` and `client`, the interaction strategy attempts to integrate each of the nonlocal KALA transactions of the `client` into the `master`. The pairwise integration of two KALA transactions is, in the INR strategy, performed according to the structure of the grammar of the KALA language defined in LTS. The grammar of a language defined in LTS is available by introspecting the composition of the linglets of a language. A KALA transaction is defined by a KALA transaction declaration, which in turn contains names, aliases, terminates, groupadds, groupaliases, groupterminates, autostart statements and significant events (begin,abort and commit). These significant events themselves contain names, aliases, terminates, groupadds, groupaliases, groupterminates, dependencies, views and delegations statements. As a consequence, integration starts at the top AST nodes representing two KALA transactions. It first tries to detect a correspondence between both top AST `Tx_Declaration` nodes by invoking the `corresponds:` method. If the declarations correspond, the two are com-

bined using the `combine:` method. In the combination that follows, the interaction strategy integrates the `name`, `alias`, `terminate`, `groupAlias`, `autostart`, `groupAdd`, `groupTerminate` and `Event` parts. Again it first tries to detect a correspondence between a part of the two declaration nodes. When a correspondence is detected, the parts are combined by invoking the `combine:` method on these parts. As the events (named `begin`, `commit` and `abort`) having the same name correspond, the resulting combination in turn triggers the integration process for their parts e.g. the `name`, `alias`, `terminate`, `groupAdd`, `groupAlias`, `groupTerminate`, `dep`, `view` and `del`.

INR strategy with deep integration in the KALA language specification

We add the INR and the Deep strategy to the KALA language specification (see below) in line 1. This line ensures that each linglet of the KALA language delegates to the interaction strategies, and as such inherit the integration mechanism defined in the INR strategy. The methods defined and provided by the `INR` strategy for each linglet of the KALA language are depicted in layer 2 of Figure 7.4.

In layer 1 of Figure 7.4 some of the linglets of the KALA language that further specialize the strategy are depicted. The interaction strategy is specialized by overriding correspond (`corresponds:` method) and combination (`combine:`) methods in the different linglets. As such, the interaction strategy knows which particular nodes of the KALA language correspond to other nodes and how they need to be combined. Linglets that do not specialize the `corresponds:` and `combine:` do not correspond with other nodes and are thus not combined, but simply added to a suitable node of the target AST by the INR strategy.

```
(1)   Linglet=Deep
         extends: INR.

(2)   Tx_Declaration
(3)      corresponds: peer {
(4)             (ast correspondType: peer)
(5)             and: [ (ast contains: 'name') ifTrue: [
(6)                       peer signature = ast signature
(7)                    ] ifFalse: [ true ] ]
(8)         }
(9)      combine: peer {
(10)             ast body do: [:child | child integrate: peer  ].
(11)             (peer contains: 'signature') ifFalse: [
(12)                peer signature: ast signature
(13)             ]
(14)          }.

(15)   Event
(16)      corresponds: peer {
(17)             (ast correspondType: peer)
(18)             and: [ ast name linglet type
```

```
                                    = peer name linglet type ]
(19)        }.

(20)   Naming=Linglet
(21)      corresponds: peer {
(22)            (ast correspondType: peer)
(23)            and: [ ast name = peer name  ]
(24)        }
(25)      combine: peer {
(26)            (ast key = peer key) ifFalse:[
(27)                  Exception raiseSignal: (ast linglet type,
                                               ' is duplicated')
(28)            ]
(29)        }.

(30)   Alias
(31)    extends: Naming.
(32)   Name
(33)    extends: Naming.
(34)   GroupAlias
(35)    extends: Naming.
(36)   GroupAdd
(37)    extends: Naming.
(38)   Terminate
(39)    extends: Naming.
(41)   GroupTerminate
(42)    extends: Naming.
```

The following INR specialization has been made:

1. KALA transactions are defined by the `Tx_Declaration` linglet. This linglet defines two parameters: `body` and `signature`. The former contains the statements of a KALA transaction declaration and the later the method signature of a KALA transaction declaration. By overriding the method `corresponds: peer` we define when a `Tx_Declaration` node corresponds with a given peer node. In line 4 we state the linglets of the peer and the current ast node correspond if the types of their linglets correspond. This is checked with the `correspondsType:` method. In addition, we also need to take into account the method signature of KALA transactions and whether or not KALA transactions have a method signature. If the current `ast` node has a method signature (stored in the `signature` part) (line 5) then the method signature of the `ast` and the peer must be the same (line 6). If the current `ast` node does not have a method signature, then the current ast node corresponds to any other transaction (line 4). Whether an AST node has a value can be checked by the LMOP method `contains:label`.

   Upon combination of the `peer` and `ast` nodes, each part of the `ast` node is

integrated into the peer node. At line 10 each child of the `body` of the ast node is integrated in the peer node by recursively invoking the `integrate:peer` method on the child. At line 11 the signature of the peer is only changed if the peer does not have a method signature.

2. Significant transaction events such as `begin`, `abort` and `commit` are defined with the `Event` linglet. The name parameter of the `Event` linglets for the different significant events, contain the linglets `"begin"`, `"abort"` and `"commit"` respectively (see Section C.2). Two `Event` AST nodes correspond if the types of their linglets correspond and if their name linglets are the same. This condition is encoded by specializing the `corresponds: peer` with a boolean expression in line 16. The first part of the boolean expression determines whether the types of the linglets of the nodes correspond (line 17). The second part determines whether the types of the linglets of their names correspond (line 18). The combination of the two `Event` nodes is completely handled by the INR strategy.

3. The `Alias`, `GroupAlias`, `Name`, `GroupAdd`, `Terminate`, `GroupTerminate` linglets each consist of a name and a key parameter. The name denotes the variable which is either used or declared, and the key is used to identify the transaction. In order to detect erroneous merges, nodes of these linglets correspond if nodes have the same name. However, they fail to combine if their expressions, used to identify the transactions, are different. As such, we can avoid erroneous merges.

   Because the `Alias`, `GroupAlias`, `Name`, `GroupAdd`, `Terminate`, `GroupTerminate` linglets all have the same combination and correspondence behavior, we encode it in a new linglet called the `Naming` linglet. This new linglet is defined at line 20 and specializes two methods `corresponds: peer` and `combine:peer` of the INR strategy. The former defines that two naming linglets correspond if their linglets correspond, and if they have the same name (lines 21-23). The latter defines that two naming linglets combine if their keys, which contain the expressions they use to identify linglets, are the same (lines 26-28).

   The common combination and correspondence behavior in the `Naming` linglet is shared by installing the `Naming` linglet as a common delegate for the `Alias` (line 30), `Name` (line 32), `GroupAlias` (line 34), `GroupAdd` (line 36), `Terminate` (line 38), `GroupTerminate` (line 41) linglets.

To conclude, in LTS, the separately defined INR strategy can be added to combine the different produced KALA transactions, without having to break the modularization of the linglets producing these KALA transaction (fragments). More so, the interaction strategy is reusable and can be used in many different language implementations (see next increment in Section 7.5).

7.5 Second Increment : Sagas

In the second increment of our family of DSTLs we implement a DSTL for the Saga transaction model we introduced in Section 7.1.1 which we refer to as the Saga DSTL.

This section follows a structure similar to the structure we used to discuss the previous language.

7.5.1 Sagas ATMS

The Sagas ATMS does not focus on the nesting depth like nested transactions, but on cooperations among sibling transactions on a single level. Sagas is designed to support long-lived transactions. These transactions are split into a sequence of atomic sub-transactions that should either be executed completely or not at all. Splitting long-term transactions releases intermediate results at an earlier stage. First, it increases concurrency as other transactions can execute concurrently. Second, it also decreases the probability of deadlocks, since after each sub-transaction all data is released. Finally, each sub-transaction will probably require fewer resources than the complete Saga.

Extra work needs to be done in order to rollback a Saga: compensating methods must be provided to potentially undo the effects of already committed sub-transactions. Hence, the application programmer should for each sub-transaction define a compensating transaction, which performs a semantical compensation action.

7.5.2 Saga DSTL by Example

Sagas introduce several new concepts: a long-lived transaction called a Saga, atomic sub-transactions called steps and compensating transactions called compensating steps or compensates.

Case

Consider the following skeleton of a banking application:

```
class Cashier {
  public void moneyTransfer (Account from, Account to, int amount){
    this.transfer(from, to, amount);
    this.printReceipt(from, to, amount);
    this.logTransfer(from, to, amount);
  }
  private void transfer (Account from, Account to, int amount)
  { ... }
  private void printReceipt(Account from, Account to, int amount)
  { ...}
  private void logTransfer(Account from, Account to, int amount)
  { ... }
}
```

The money transfer method is conceptually composed of three actions: first perform the actual bank transfer, second print a receipt and third update the logs of the bank. The first action is the transfer itself, which updates both accounts involved in the bank transfer. The second action prints out a receipt which the cashier hand over to the

customer as proof of the transfer method. The third and last step updates the logs of the bank to reflect that a transfer has been performed and a receipt has been given.

Conceptually, the money transfer method is one atomic event, and needs to be made transactional. Implementing the entire money transfer method as one transaction however is problematic. This is because printing out the receipt takes a few seconds even on a fast printer. This effectively turns the money transfer method into a long-lived transaction.

The Sagas ATMS suits the money transfer method better as it turns the long-lived transaction into a series of steps. Each action of the transfer maps to a step in a Saga, and for each step a compensating step is possible. The compensating step of the transfer itself is a transfer in the opposite direction. The printout of the receipt can be compensated by printing out a transfer cancellation notice. The last step of a Saga has no compensating step, as the Sagas ATMS does not require this.

Equivalent KALA code

The KALA specification for the above Saga example is quite large and verbose. We refer the reader to Section C.3.1. Section 7.1.2 shows the KALA equivalent of the second step of the saga example and explains it in detail. We discuss the remaining part of the KALA specifications, when we implement the Saga DSTL in LTS.

DSTL

A Saga is declared by the keyword `Saga`, followed by the method signature of the method executing the Saga, and a list of steps. Each step is an atomic sub-transaction declared by the keyword `step`. All steps, except for the last one, require a compensating step. This is declared by the keyword `compensate`, followed by the signature of the method executing the compensating step. The compensation of a step is invoked when a step has to rollback. As such, we need to specify arguments for the parameters of the method declared as a compensate. The list of arguments is prefixed by the keyword `params` and enclosed in `<` and `>`. Some of these arguments are local variables of the step method, because the compensating transaction executes in another thread which is spawned before the step has even begun to execute. An indirection is used to implement this. For this, in the Saga DSTL, the local variables which must be passed to the compensating step have to be declared explicitly by the keyword `wrap`.

The code below shows an example of a Saga declaration containing three steps. Each step is accompanied by a compensating step which takes a list of parameters and an optional declaration of wrappers.

```
Saga Cashier.moneyTransfer(Account, Account, int) {
   step Cashier.transfer(Account, Account, int)
      compensate transfer(Account, Account, int)
      params <to, from, amount>
   step Cashier.printReceipt(Account, Account, int)
      compensate printTransferCancel(Account, Account, int)
      params <from, to, amount, num_receipt>
```

```
      wrap(num_receipt)
   step Cashier.logTransfer(Account, Account, int)
}
```

The `moneyTransfer(...)` method is the top level transaction of a Saga. This transaction lists the steps which are executed as sub-transactions. The first step is the `transfer(...)` method, with as compensating step also the `transfer(...)` method, in which source and destination accounts have been swapped. The second step is the `printReceipt(...)` method, which is compensated by the `printTransferCancel(...)` method. The local variable `num_receipt` of the `printReceipt(...)` method is passed via the `wrap` declaration to the compensating step. As such, the compensating step can include the receipt number in the cancellation notice. The last step of the Saga is the `logTransfer(...)` method which, as explained, has no compensation.

7.5.3 DSTL Translational Semantics

The translational semantics of the Saga DSTL is depicted in Figure 7.5. The first concept is a "top level" Saga transaction ((1) in the figure), the second is a step and its compensating step ((2a) en (2b) in the figure). We now discuss each of the concepts in more detail:

Saga The translational semantics of the Saga concept in line 1 of Figure 7.5 consists of a list of KALA transaction specifications containing: the Saga transaction, and a transaction for each of its steps.

Steps and Compensating Steps The translational semantics of a step and its compensating step is a complex KALA program ((2) in Figure 7.5). The complexity is due to three phenomena we observe:

Scattering of KALA code The DSTL program and its equivalent KALA specifications are marked with colors. The code marked in red corresponds to a top-level Saga and its translational semantics. The code marked in yellow concerns a step of a Saga and its translational semantics. The code marked in green is about a compensation of a step and its translational semantics.

The code marked in yellow ((1) in Figure 7.5) indicates that the semantics of a `step` clause invasively changes the semantics of its top-level Saga transaction. In addition, the KALA equivalent marked in yellow of a step and its compensating clause ((2a) en (2b) in Figure 7.5) indicates that the semantics of a step clause additionally produces its own KALA transaction. In other words, the semantics of a `step` clause comprises several code KALA code fragments. We recognize this phenomenon as an instance of multiple results (P3).

Similarly, there is code marked in green in the KALA equivalent of a step ((2a) en (2b) in Figure 7.5), which is the equivalent of a `compensate` clause.

Hence, the semantics of a `compensate` clause invasively changes the semantics of a step clause. We also recognize this phenomenon as an instance of multiple results (P3).

Duplicate KALA code There is one KALA statement which is both marked in yellow and green because it part of the semantic equivalent of both the `compensate` and the `step` clause. Naturally, upon integration of the KALA fragments such duplicate KALA statement needs to be pruned.

Source program queries The translational semantics of a step and its compensating step comes in two variants. If a step is not the first one, then additional KALA statements are necessary to ensure the correct rollback of the compensating steps of its Saga in reverse order (2a). If a step is the first one, then there is no previous compensating step to rollback (2b). Hence, upon translation of the `compensate` clause, the source program needs to be queried to determine if a step is the first in its Saga or not. We recognize this phenomenon as an instance of multiple inputs (P2).

It is clear, that the semantics of the various DSTL clauses is not a straightforward one-to-one mapping. The challenge is to be able to define the semantics such that the semantical specifications can be easily composed and used without requiring invasive changes to their definitions.

7.5.4 Overview of the Implementation in LTS

In the Saga DSTL we distinguish between three language constructs: the Saga declaration (`saga` clause), the atomic sub-transaction or step (`step` clause) and the compensating transaction (`compensate` clause). Steps and compensating steps are considered as two different language constructs because of several reasons. First, as we have explained earlier, compensating steps are optional: all steps, except for the last one, have compensating steps. Second, compensating steps are the mechanisms offered by the Saga ATMS to undo, in case of a rollback, the work performed by a single step. In principle, other ATMS can offer other mechanisms to rollback steps. Therefore, a step is a concept which is, in fact, independent of the notion of a compensating step of a Saga ATMS. The arguments used to call the compensating steps and the declaration which of these arguments are local variables of the transactional method executing the step, are not considered as separate constructs but are considered part of the `compensating` construct. The three language constructs define a top-level Saga, steps and compensating steps for these steps respectively. In LTS, each language construct is implemented in a linglet. As a consequence, a Saga declaration is defined by the `Saga` linglet (see Section 7.5.5), a step is defined by the `Step` linglet (see Section 7.5.6) and a compensating transaction is defined by the `Compensate` linglet (see Section 7.5.7).

The interactions among constructs are not a part of the linglets that define them. More precisely the combination and integration of the KALA fragments are implemented with the INR and the source query to determine whether a step is the first one is implemented with a new interaction strategy called the Sibling strategy. The INR strategy extends the LMOP such that the linglets are not polluted with combination

and integration semantics. The Sibling strategy is an application-specific interaction strategy, which is designed to further reduce the coupling between the compensating linglet and the other linglets. As such we preserve the modularity of language constructs.

The Saga DSTL is defined in a language specification (see Section 5.4) in which its linglets are combined, in which the LTS system is extended with interaction strategies and in which the interactions among linglets are implemented with the Sibling and INR strategies (see Section 7.5.9 and Section 7.5.10).

We do not explain the modularization of the language constructs of the Saga DSTL by their corresponding linglets in detail, but only emphasize the new challenges that this increment raises.

7.5.5 The Saga Language Construct

The Saga language construct which is implemented by the `Saga` linglet, is shown below.

```
Linglet Saga {
   syntax { "Saga" (package ".")* class "." method
                "(" !(parameter ("," parameter)* ) ")"
                "{" child* "}" }
   generate{ | package class method parameter children toplevel |
      package := ast package generate.
      class := ast class generate.
      method := ast method generate.
      parameter := ast parameter generate.
      children := ast child generate.
      toplevel := #Tx_Declaration {
         'package.'class.'method('parameter) {
            name(self <Thread.currentThread()>)
         }
      }.
      ASTLoseSet add: toplevel add: children
   }
}
```

The syntax of the `Saga` linglet consists of a keyword `Saga` followed by a method signature (denoted by the parameters `package`, `class`, `method` and `parameters`) of a Saga and zero or more children (denoted by `child*`) which are enclosed in parentheses. Observe that the `Saga` linglet has a multivalued syntactical parameter called `child` which is later, in the language specification, bound to the `Step` linglet. This is similar to the `Tx` linglet discussed in Section 7.3.2. Note these two linglets could be refactored such that their common syntax that defines a method signature could be specified in a separate linglet.

The `generate` method of the `Saga` linglet defines the translational semantics of a Saga in full accordance with line 1 of Figure 7.5. More precisely, the `Saga` linglet produces the code marked in red. It returns a set of transaction specifications: the

`toplevel` transaction specification of a Saga, and the atomic sub-transaction specifications defined by each `child` part. The two results `toplevel` and `children` are returned as an `ASTLoseSet`[5].

The only behavior specified by the `toplevel` transaction is that it registers itself (denoted by the KALA pseudovariable `self`) using the current thread object with the KALA `name` statement. As such, the atomic sub-transactions defined by steps can refer to their toplevel Saga transaction. We refer to Section 7.1.2 for the rationale behind this.

7.5.6 The Step Language Construct

Steps are the atomic sub-transactions of Sagas, and are defined by the following `Step` linglet.

```
(1)     Linglet Step {
(2)       syntax { "step" (package ".")* class "." method
(3)                      "(" !(parameter ("," parameter)*) ")"  }
(4)       generate { | package class method parameter res |
(5)         package := ast package generate.
(6)         class := ast class generate.
(7)         method := ast method generate.
(8)         parameter := ast parameter generate.
(9)         res := #Tx { 'package.'class.'method('parameter) {
(10)           alias (Saga <Thread.currentThread()>)
(11)           groupAdd (self <""+Saga+"Step">)
(12)           begin {
(13)             dep(Saga ad self, self wd Saga)
(14)           }
(15)         }.
(16)         res nonlocals add:
(17)           #Commit {
(18)              commit { groupTerminate(<""+self+"Step">)
(19)              terminate(self) } }
(20)           role: #terminate.
(21)         res nonlocals add:
(22)           #Abort {
(23)              abort { groupTerminate(<""+self+"Step">)
(24)              terminate(self) } }
(25)           role: #terminate.
(26)         res.
(27)       }
(28)     }
```

[5]An ASTLoseSet does not create an AST, but merely acts as container to hold the AST nodes. The set is also normalized i.e. a set of sets is flattened.

Syntax

The `Step` syntax is defined at line 2 consisting of the keyword `"step"` and a method signature (denoted by the parameters `package`, `class`, `method` and `parameters`).

Modularizing semantics: Multiple Results

The translational semantics is defined in full accordance with the semantics depicted in lines 1, 2a and 2b of Figure 7.5. The `Step` linglet produces three results: a new transaction (line 4) containing the KALA statements marked in yellow of lines (2a) and (2b) of Figure 7.5, a `commit` statement (line 17) and an `abort` statement (line 22) which are the KALA statements marked in yellow in line (1) of Figure 7.5.

The first result produced by the `Step` linglet is a transaction which declares a given method as a sub-transaction of a Saga (line 9). The produced step transaction, establishes some dependencies with its top-level Saga. Therefore its top-level Saga transaction is looked up via the `alias` statement using the current thread object as the name to identify the transaction (line 10). At the beginning of a step, a couple of dependencies are set so that if a step aborts, its top-level Saga necessarily abort as well (`ad` dependency line 13), and so that if its top-level Saga aborts, the step aborts as well (`wd` dependency line 13). The step transaction also adds itself (`self` pseudovariabele in KALA) to a group which is identified by the expression `""+Saga+"Step"` (line 11). This expression yields a unique group name for each Saga per thread. This group name is used in the `commit` (line 17) and `abort` statements (line 22) which terminate all transactions in that group when their top-level Saga transaction aborts (line 23) or commits (line 18).

The second and third produced results are the `commit` and `abort` statements. They need to be integrated in the transaction specification defined by the `Saga` linglet. However, due to the isolation of linglets, the `commit` and `abort` statements can only be defined (see Section 4.8), not directly integrated. In LTS, such results are considered nonlocal results or plainly nonlocals. It suffices to hook the `commit` and `abort` statements to the step transaction as nonlocals (lines 16 and 21). The integration of the nonlocals has to be defined in the language specification (see Section 7.5.10 on page 305). The two nonlocals are given a role name, so that in the language specification the language designer can encode behavior to handle the nonlocals. As both nonlocals need to be treated similarly, they both play the role `#terminate`.

7.5.7 The Compensate Language Construct

Compensating transactions are defined by the `Compensate` linglet, which is shown below.

```
Linglet Compensate {
  syntax { "compensate" (package ".")* class "." method
                        "(" !(parameter ("," parameter)* ) ")"
         !( "params" "<" argument ("," argument)* ">" )
         !( "wrap" "(" localvariable ("," localvariable )* ")" )
```

```
}
generate { | package class method parameter txmethodsignature
             arguments localvariables prevdep prevalias |
  package := ast package generate.
  class := ast class generate.
  method := ast method generate.
  parameter := ast parameter generate.
  txmethodsignature := ast txmethodsignature generate.
  arguments := ast argument generate.
  localvariables := ast localvariable generate.
  prevdep := ast prevdep.
  prevalias := ast prevalias.
  #Tx_Declaration {
      'txmethodsignature {
      alias (Saga <Thread.currentThread()>)
      'prevalias
      autostart ('package.'class.'method('parameter)
                 <'arguments>
                 ('wraps) {
                 name(self <""+Saga+"Comp">)
                 groupAdd(self <""+Saga+"Comp">)
      })
      begin {
                 alias (Comp <""+Saga+"Comp">)
                 dep(Comp bcd self)
      }
      commit {
                 alias (Comp <""+Saga+"Comp">)
                 dep(Comp cmd Saga, Comp bad Saga)
                 'prevdep
      }
  }
}
prevalias {
  (ast previous == nil) ifFalse: [
      #Alias{
         alias (CompPrev <""+Saga+"Comp">)  }
  ] ifTrue: [ '' ].
}
prevdep {
  (ast previous == nil) ifFalse: [
      #Dependency{
         dep(CompPrev wcd Comp) }
  ] ifTrue: [ '' ].
}
```

```
  previous { }
  txmethodsignature { nil }
}
```

The `Compensate` linglet consists of six methods: the `syntax` method defining the syntax, the `generate` method defines the translational semantics, the remaining methods `prevdep`, `prevalias`, `previous`, `txmethodname` are auxiliary methods for computing the translational semantics. We do not explain the KALA semantics of compensating steps in full detail. Instead, we refer the reader to Section 7.1.2.

Syntax

The syntax consists of a keyword `"compensate"`, the signature (denoted by the parameters `package`, `class`, `method` and `parameters`) of the compensating method, and two optional clauses to specify the arguments to be used to call the compensating method (prefixed by the keyword `"params"`) and to specify which of those arguments is a local variable (by keyword `"wrap"`).

Modularizing semantics: scattering of results

Figure 7.5 shows that the semantics of the `compensate` clause invasively changes the semantics of a step. In LTS, we are able to modularize the semantics of the `compensate` clause such that it is defined independently from the semantics of the `step` clause (defined in the `Step` linglet). The `Compensate` linglet just produces a new compensating transaction declaration, which we refer to as the *compensating transaction declaration*, containing additional transactional properties that are needed to turn an atomic sub-transaction of a Saga (produced by the `Step` linglet) into a transaction with a *compensating transaction* which is the transactional method that actually compensates the work performed in a step in case of a rollback.

As a consequence, the translational semantics of the `Compensate` linglet does not need to compose its translational semantics with the semantics of the step. An interaction strategy will be used to realize the interaction between the `Compensate` and the `Step` linglet. Hence, we effectively preserved the modularization and isolation of the `Step` linglet despite its invasive semantics.

Modularizing semantics: partial results

The `Compensate` linglet has several parts, which are used to define its translational semantics. The method signature (`txmethodsignature`) of the compensating transaction declaration is not defined in the `Compensate` linglet. In LTS, the need for this additional information is stated by declaring the method `txmethodsignature{...}`. It returns nil, and as a result renders the compensating transaction declaration a partial result: it does not have a method signature. The method signature is non-essential information, which may be, in full accord with the model requirement (R1a), omitted by the linglet. As such, we further modularize the `Compensate` linglet as its semantics does not need to concern itself with the method signature of the compensating transaction declaration.

Modularizing semantics: Multiple Inputs

The translational semantics of the `Compensate` linglet contains both case (2a) and (2b) of Figure 7.5 which depend on whether its step is the first one or not. If its step is not the first one, then additional KALA statements are necessary to ensure the correct order of execution of the compensating steps of its Saga. These additional KALA statements are computed by two auxiliary methods `prevdep` and `prevalias`. The `prevdep` method computes the dependency on the previous step of its Saga, the `prevalias` method computes the alias for identifying the previous step of its Saga. Both these methods need to determine if there is a previous step. Whether there exists a previous step, is not defined by the `Compensate` linglet. In order to compute it, the source AST must be traversed. Hence, the translational semantics of the `Compensate` linglet requires multiple inputs: the information which is declared by its syntactical definition and whether there exists a previous step. The computation based on source AST traversals to determine whether there is a previous step cannot be a part of a linglet as it would break the modularity of the `Compensate` linglet. Therefore we declare the need for this additional input as an empty method. As such, the linglet can then use this information as if it were present.

In contrast to the `txmethodsignature` method, an implementation for the `previous` method has to be provided, because whether a step is a first one is essential to be able to compute the semantics the `Compensate` linglet. The need to provide an implementation for this method upon the use of the `Compensate` linglet in a language specification is ensured in LTS as empty methods are considered abstract methods.

7.5.8 The Entire Saga Language

The combination of the `Saga`, `Step` and `Compensate` linglets for constructing the Saga DSTL is shown in the language specification below.

```
(1)    name Saga
(2)    base KALA

(3)    Linglet=Sibling extends: SSQ.

(4)    Top=Saga
(5)             package: ID.
(6)             class: ID.
(7)             method: ID.
(8)             parameter: ID.
(9)             child: Compensated. Step.

(10)    Compensated=Concat
(11)            left:Step.
(12)            right:Compensate.

(13)    Compensate
```

```
(14)            package: ID.
(15)            class: ID.
(16)            method: ID.
(17)            parameter: ID.
(18)            argument: Expression.
(19)            localvariable: ID.
(20)            previous: {
                   ast leftsibling
(22)            }.

(23)    Step
(24)            package: ID.
(25)            class: ID.
(26)            method: ID.
(27)            parameter: ID.
(28)            nonlocalterminate: el : {
(29)               el integrable: master {
(30)                   master source
(31)                   == (ast source
(32)                           ancestor: 'Top') }  }.
```

The above language specification also has to implement two interactions among its linglets. One interaction queries the source program for retrieving the previous step required by the `Compensate` linglet. The other interaction integrates the `terminate` nonlocals produced by the `Step` linglet in the correct KALA transaction. The interactions for the `Compensate` and `Step` linglet are implemented with a suitable interaction strategy by extending them. The previous step (line 20), computed in the `previous` method, is retrieved by using a new interaction strategy. The nonlocals with the role `terminate` (line 28) are relocated by the logic implemented in the `nonlocal`*`terminate`*`:el` method. In the next two subsections each of those interactions is discussed in more detail.

In addition, there is a third interaction between the `Step` and the `Compensate` linglet for combining their produced transaction fragments in one fragment.

7.5.9 Application-specific Interaction Strategies

In this section, we device a new interaction strategy called Sibling which is specifically designed for this case study. The new interaction strategy allows us to further decouple the `Compensate` and other linglets in the source program query that retrieves the previous step.

Why do we need a new interaction strategy?

There are several ways to retrieve the previous step (line 20 in the language specification). The most straightforward way is to implement a query with the SSQ strategy, which is shown below.

```
previous: { | saga currchild indexchild |
   saga := ast ancestor: 'Saga'.
   currchild := ast up:
                  [ :child | child parent linglet type = 'Saga' ].
   indexchild := saga body indexOf: currchild.
   (indexchild == 1)
      ifTrue: [ nil ]
      ifFalse: [ (saga body at: (indexchild -1))
                       descendant: 'Compensate' ]
}
```

This algorithm is quite simple. We locate the ancestor `Saga` node of the current compensate node by ascending the source AST. We then check if there is a sibling in the `Saga` node. The challenging part of the algorithm is that the `Compensate` node is not a direct part of the `Saga` node, but nested several levels down a subtree of some part of the `Saga` node (see Figure 7.6). Therefore we need to find the part of the `Saga` node which the `Compensate` node belongs to. We do this by ascending from the current `Compensate` node up to a node which is a part of the `Saga` node. The `Compensate` sibling node is the descendant of the sibling of the found part of the `Saga` node. Observe that we rely on the `descendant:` method of Structure Shy Query (SSQ) strategy for retrieving the `Compensate` sibling node.

The above implementation is quite structure shy as we do not detail exactly how deep the `Compensate` node is located in the AST. However, the query does hard-code the point where multiple steps can occur i.e. the `Saga` node. To further reduce this explicit coupling between the `Compensate` and the `Saga` linglet, we designe a new interaction strategy called the Sibling strategy.

Sibling Strategy

The Sibling strategy extends the LMOP with the method `leftsibling`. This method searches the AST tree upwards until an AST node is found in which multiple branches can be stored. As such, there is no explicit need to hard-code the `Saga` node in the implementation of the interaction. The search for a sibling using this interaction strategy is reduced to a call to the method `leftsibling`.

In Figure 7.6, we depicted the execution of the Sibling strategy for the second compensating step of our example Saga (given at the beginning of Section 7.5). The black tree is the source AST of our Saga example implemented in LTS. The search starts at the `Compensate` node, ascending the source AST. The first parent is the `Concat` AST node. This node has two parts `left` and `right`, both of them containing a single AST node. Hence the search is continued upwards in the tree. The second parent is the `Saga` node whose has one part called `child` which can contain multiple AST nodes. Hence, the search is stopped because we found an AST node where multiple branches can be stored. Finally we retrieve the left sibling by descending form the `Saga` node down to the sibling `Compensate` node.

Implementation of the Sibling Strategy

The code below merely shows a snapshot of the `Sibling` strategy sufficient for the purpose of this dissertation. The core of the interaction strategy is implemented in the `_findpart:` method. It is a recursive method that checks whether the current node is stored in a multivalued slot of its parent i.e. a slot which contains multiple AST nodes. In order to implement this we rely on the introspective capabilities of the LMOP for traversing the source AST and for determining whether an AST node can, according to the grammar, have multiple AST node parts.

Recall that AST nodes store their parts in slots (see Section 5.3.2). Slots are identified by names which we call labels. A slot can either contain a single AST node or a set of AST nodes. The `Sibling` strategy retrieves the list of slots by using the method `members`, and accesses the value of the slots by using the method `member:label`.

The `_findpart:` method is implemented as follows. First, the `_findpart:` method searches for the label of the slot in a given `parent` in which the current node is stored. Second the method `multivalued` of the LMOP is called in order to determine whether the found slot can contain multiple AST nodes. If that is the case, and if the current node is not the first child, the previous child is returned.

Once we obtained the sibling part of the `Compensate` node within the `Saga` node, we descent in (the subtree denoted by) this part down to a `Compensate` node. The descend is performed by the `descendant:type` method of Structure Shy Query (SSQ) strategy (see Section 6.3.1). The method uses the given type parameter until a node is found whose linglet has that given type. Observe that the Sibling strategy does not hardcode the type of the `Compensate` node. Instead the type of the linglet of the descendant is the type of the current node which initiated the search of a sibling. The linglet and its type can be accessed using the LMOP methods `linglet` and `type`.

```
1  Linglet Sibling {
2    leftsibling { | part |
3       part := ast _findpart: ast parent.
4       (part == nil) ifTrue: [ nil ]
5                     ifFalse: [ part descendant: ast linglet type ]
6    }
7    _findpart: parent { | label index |
8       label := parent members detect: [ :label |
9         index := 0.
10         (parent member: label) detect: [ :child |
11             index:= index + 1. child == ast
12          ]
13       ].
14       (parent linglet multivalued: label) ifTrue: [
15         (index == 1)
16              ifTrue: [ nil ]
17              ifFalse: [ (parent member: label) at: (index-1)  ]
18       ] ifFalse: [
19          parent _findpart: parent parent
20       ]
21    }
22 }
```

Deployment and use of the Sibling strategy

The Sibling strategy is deployed in line 3 of the language specification. This line ensures that each linglet of the Saga DSTL delegates to the Sibling strategy, and as such inherits the mechanisms defined in this interaction strategy. This is necessary because the Sibling strategy is defined as an extension of an interaction strategy supporting the `descendant:` method (line 5 of its definition). That method is defined in the SSQ strategy.

The `previous` method provided by the language specification at line 20 is implemented by using the `leftsibling` method of the Sibling strategy.

7.5.10 Source-steered Integration

In this section, we show that the multiple results produced by the `Step` linglet get integrated using the source language (SP8) using the INR strategy. More so, for the integration of these results the source language is indispensable as the produced KALA transaction fragments lack the intentions required for the integration of the KALA fragments.

Recall from Section 7.5.6 that the `Step` linglet produces two nonlocal `terminate` statements. Both nonlocals have the same role, namely `terminate`. The nonlocals are `terminate` statements which can be integrated in any transaction, according to the

KALA grammar which is used by the INR strategy to determine whether or not a nonlocal can be integrated.

Saga declarations are translated into many transaction specifications: Recall that each Saga declaration is translated to a top level transaction and so is each of its steps. Moreover, there can be many Saga declarations. This fact renders the integration based on the grammar of the nonlocal ineffective, because the nonlocal terminate statement must be integrated in a specific transaction. Informally put, the `terminate` statements must be integrated in the top level transaction of the Saga which contains the producing `Step` linglet.

In order to integrate the nonlocal `terminate` statements into the proper transaction specification, we tweak the INR strategy for these nonlocals. Recall that the INR strategy is an incremental process, successively attempting to integrate nonlocals in an AST node (called the `master`) of the current target program AST. The interaction strategy uses the `integrable: master` method to decide whether or not nonlocals can be integrated in a given node of the target program. By specializing the `integrable: master` method on the nonlocal (line 29 in the language specification), we ensure that the nonlocals are integrated in the correct transaction. This overriding is performed in the `nonlocal`*`terminate`*`:master` method of the metaobject protocol (line 28). The method is executed upon the production of a nonlocal playing the role `terminate`.

In the `integrable: master` method we need to identify the proper top-level Saga in which the nonlocals should be integrated. Doing this according to the target language is possible but suffers from dependencies on the semantics of the `Saga` linglet and is actually only accidentally correct. Let us return to the definition of the `Saga` linglet, to understand why this is the case. The `Saga` linglet produces a KALA transaction for a method signature. So, upon the integrable check of the nonlocal `terminate` statement, we would have to check whether the method signature of a given KALA transaction is the method signature of the `Saga` linglet containing the `Step` linglet producing the nonlocal. This check depends on the fact that the transactional properties of a method are declared by a single KALA transaction. Transactions can in principle not be identified using their method signature, as there can be several transaction specifications for the same method signature [Fab05].

Instead of using the target language to identify in which transaction the nonlocal `terminate` statements have to be integrated, we use the source language. We can basically write the informal condition of whether or not to integrate in a given `master` in a formal way: a `master` qualifies if its producing node is the `Saga` node which contains the step that produced the `terminate` nonlocals. The producing linglet can be retrieved by the method `source` of the LMOP.

Figure 7.7 graphically illustrates the source-steered integration defined at lines 29-32 of the language specification in five steps.

- In step 1, the `terminate` nonlocals are moved up in the target AST to a new AST node, and an integration of the nonlocals with that new AST node is attempted. The integration process of INR invokes the `integrable: master` method on the nonlocals (line 29).
- In step 2 at line 30, the source node of that new AST node (i.e. `master`) is

retrieved, which is a `Saga` node.

- From the source node of the nonlocals determined in step 3 at line 31, which is a step source AST node, in step 4 at line 32 a query is launched to determine its `Saga` node. The query is implemented using the `ancestor:` method of the Structure Shy Query strategy (SSQ).
- In step 5 at line 31, the `Saga` node obtained from the second step, is compared with the `Saga` node obtained from the fourth step. If the two nodes are the same, this means that the nonlocals indeed are integrated in the `Saga` node which contains the `step` node that produces the nonlocals.

Note that in order to further reduce the dependencies between the actual linglets we refer to aliases instead of the actual linglets.

7.5.11 Resolving Duplicate Code Fragments

Figure 7.5 depicting the semantics of the Saga DSTL, shows that the `Step` and the `Compensate` linglet each produce a transaction specification which needs to be combined into a single specification. The combination of transaction specifications is taken care of by the INR strategy, and has already been discussed in Section 7.4.10.

The challenging issue of combining the transaction specifications produced by the `Step` and the `Compensate` linglet is that they both contain the same `alias` statement. Duplicate nodes and thus also `alias` statements are detected by the INR protocol because upon the integration of two transaction specifications, corresponding parts are combined. In Section 7.4.10, we stated that two `alias` nodes correspond if the names of the variables they declare correspond. Furthermore, their combination succeeds if they use the same expression to lookup the transaction. Using this definition, duplicate `alias` statements correspond and are successfully combined into a single statement.

7.6 Discussion

This chapter has validated our proposed approach for modularizing the implementation of DSTLs according to their language constructs by implementing a family of domain-specific languages using a shared pool of language constructs.

The semantics of the different language constructs is not trivial. Every restriction R0 to R4 (see Chapter 4) we imposed for a proper modularization of the language constructs, is obeyed in the definition of the linglets and is later on tackled in the language specification:

R0 partial and changeable program fragments (see Section 7.4.5) and consistency of a program fragment (see Section 7.4.10)

R1 compositionality (see Section 7.4.8)

R2 multiple inputs (see Section 7.5.7)

R3 multiple results (see Section 7.5.6).

R4 higher-order grammars (in every increment, in every linglet).

Each language construct is defined as a modularized linglet. Due to the strict separation and isolation of language constructs we are able to use a linglet pool for constructing three different languages.

The isolated linglets used to construct the DSTLs are adjusted to the language context by using interaction strategies which in turn indirectly use the metaobject protocol of LTS. Generic interaction strategies defined in the previous chapter such as SSQ and INR are successfully reused in the implementation of all these languages. The flexibility of LTS for defining new interaction strategies for tackling the specific separation of concern problem at hand is put into practice in the last increment.

We have shown that our language implementation design technique effectively modularizes the implementation of individual language constructs. Although the scope of this dissertation is the modularization of language constructs, this validation also is a first step towards an evaluation of the gained benefits of such a modularization. We have illustrated the gained benefits of:

- *Understandability*: The DSTLs have complex translational semantics which challenge the modularization of language constructs: the syntax of language constructs explicitly depend on each other (P4) and, the semantics of language constructs yield partial results (P0), do not compose (P1), require multiple inputs (P2) and produce multiple results (P3). Despite that fact, the structure of the different compilers still mimics our mental picture of the languages i.e. they are constructed using linglets where each linglet defines a language construct. This facilitates the understandability of the implementation of the compilers both during construction as during future evolutions, as we are able to focus on one construct at a time. Recall that the `extends` keyword for nested transactions extends the syntax of a classical transaction with a new clause, and the semantics of that keyword invasively changes the semantics of the plain transaction. Nevertheless, the `Extends` linglet defines the syntax and the semantics of the `extends` keyword in isolation. The composition deficit (Section 7.4.9) and integration of the semantics (Section 7.4.10) is resolved in separate concerns like the language specification and strategies.

- *Evolvability:* The different languages and their implementations are easier to evolve and maintain as they co-evolve together. We have started with the individual definition of the necessary language constructs which comprise for each construct a syntactical definition to define a language and a semantical definition to define the implementation of that language. Afterwards, the constructs are combined yielding a language and compiler of this language.

- *Extendibility:* An extension of language implementations boils down to the addition of a separate linglet defining the new language construct. The SimpleNestedTx DSTL is defined as an extension of the ClassicalTx DSTL with a couple of linglets each representing a construct of the new language. Thanks to the generic interaction strategies such as SSQ and INR, and the application-specific strategy such as Sibling, the interactions among the constructs do not cripple the extensibility as they are handled by a separate concern. Moreover, strategies render the

composition of linglets more robust such that languages can be extended more easily.

- *Reusability:* Language constructs are reusable in various languages as the implementation of the language constructs is described in separate modules, which can be used to construct other languages. Indeed, language constructs like the classical transaction are reused to construct nested transactions. Syntactically, the classical transaction and the root transaction are the same, but their semantics are not entirely the same. Despite that fact, we can still use the classical transaction because we can adapt the semantics of constructs defined in linglets in order to fit them in a new language context. The generic interaction strategies establishing the interactions among constructs such as SSQ and INR are successfully reused in the implementation of all these languages.

- *Iterative development:* All these advantages combined enable developers to engage in the natural process of developing a language, i.e. incremental language and compiler codevelopment. The set of languages are constructed by subsequent iterations, starting from the small ClassicalTx DSTL all the way to the more complex Sagas.

7.7 Conclusion

Using the novel language development technique LTS presented in chapter 5, we are able to construct a family of languages by composing language constructs. We used a set of reusable linglets and a set of reusable interaction strategies based on LMOP.

The linglets of our case study each define a modularized language construct. Each definition adheres to the five requirements postulated in Chapter 4. Using this shared set of linglets, we defined three different languages: The ClassicalTx DSTL to specify classical transactions, the SimpleNestedTx language to specify nested transactions and the Saga DSTL to define Sagas.

Complex interactions among linglets for retrieving information and for combining the transaction specifications fragments are handled in language specifications, external to linglets, by interaction strategies. We relied on the SSQ and the INR strategy which are defined in Section 6.3.1 and Section 6.3.3. In addition, a new interaction strategy was designed, tailored for preserving the separation of concerns in the Saga DSTL.

As such, the developer can rely on expert knowledge to realize the interactions and can focus on the semantics of a single language construct at a time. Because interaction strategies can be tailored and specialized for the modularization challenge at hand, an optimal interaction strategy can be chosen.

We can conclude that our language implementation design technique effectively modularizes the implementation of individual language constructs. In addition, we illustrated some the gained benefits of such a modularization like understandability, evolvability, extendibility, reusability and iterative development.

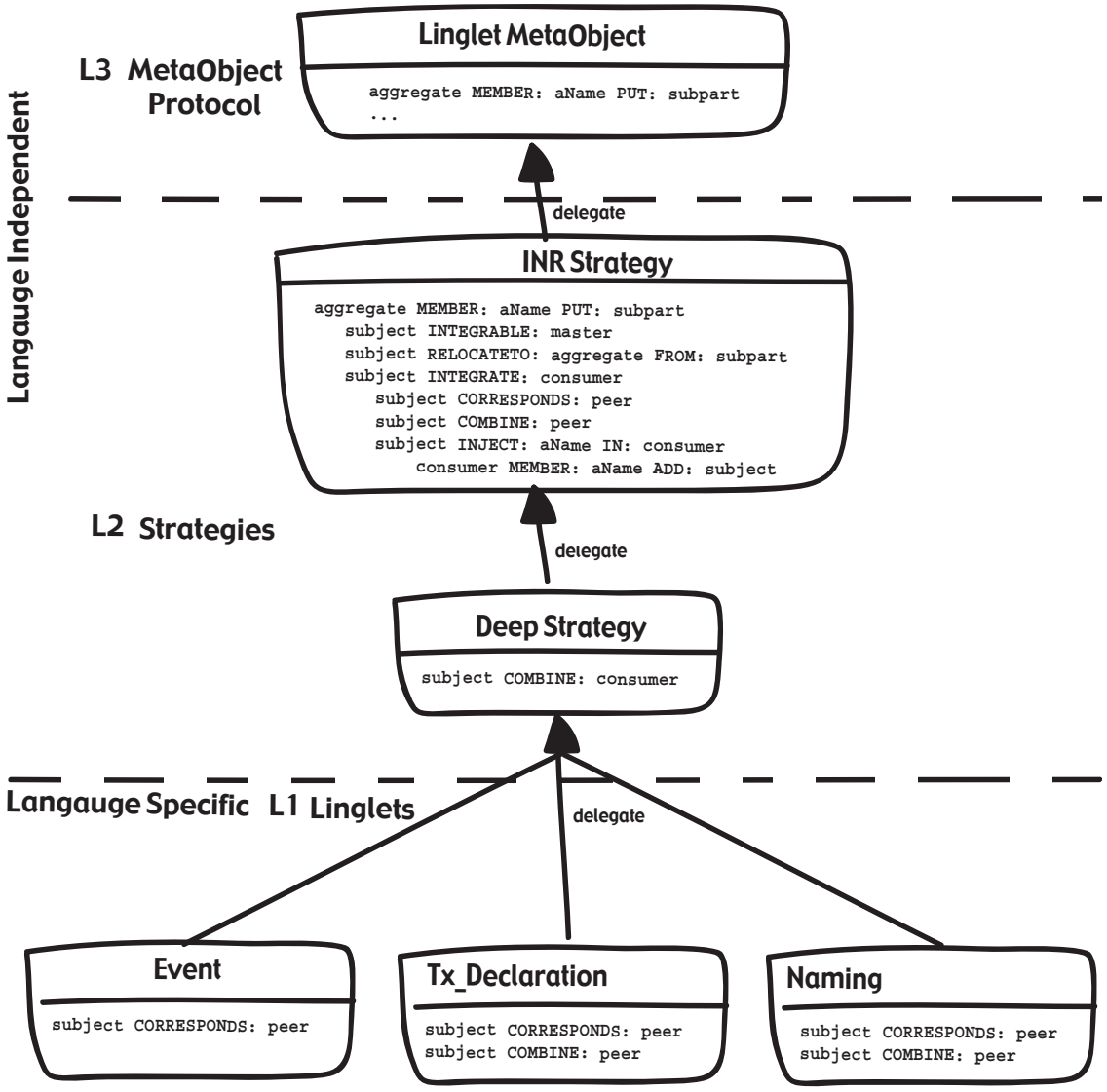

Figure 7.4: An overview of the deployment of the INR strategy in KALA.

Figure 7.5: Translational semantics of the Saga DSTL.

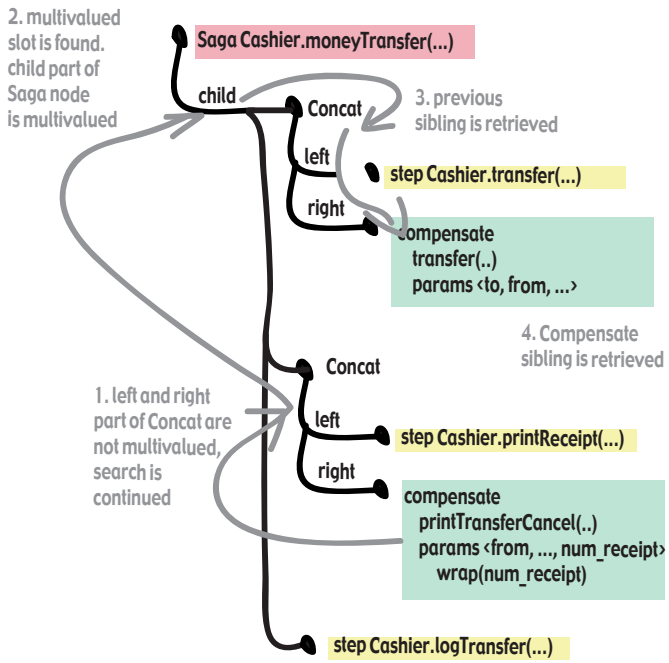

Figure 7.6: Execution of the Sibling strategy for a second compensating step of our Saga example.

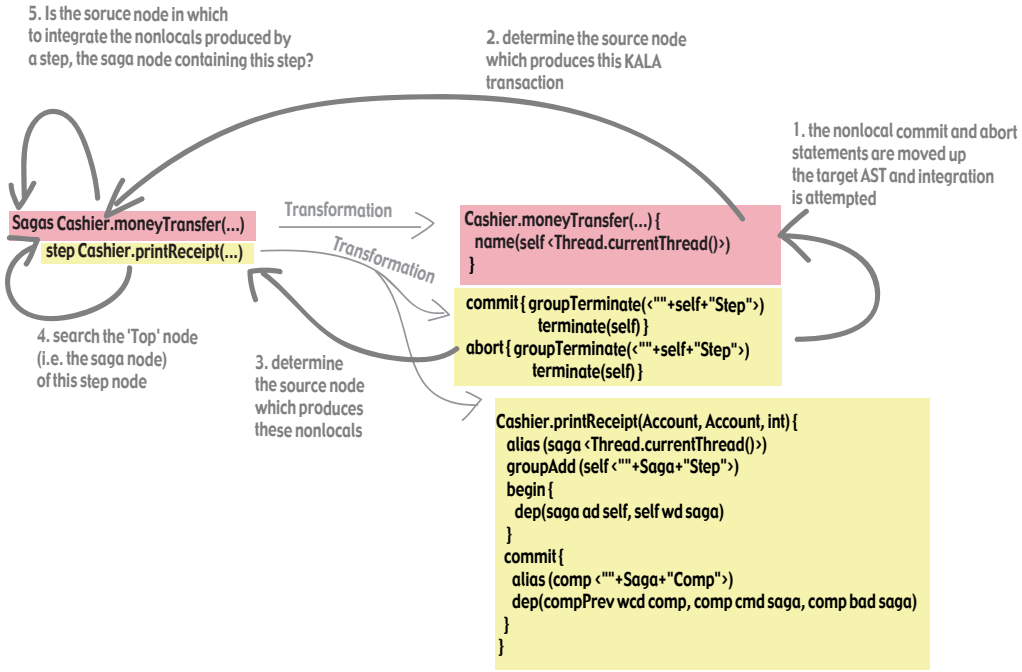

Figure 7.7: Source-steered integration of the `terminate` nonlocals.

Chapter 8

Conclusion

8.1 Research Context

Programming languages shape the way developers think and solve problems, but programming languages themselves get shaped by the increasing experience of developers. On the one hand, depending on the available language constructs, programmers reason about their problems differently. On the other hand, programming languages are grown with new constructs so that programmers can express more easily the problems from their domain within the language they are using. There is thus a continuous interaction between programming languages and the programs that developers try to express. We identify this continuous interaction by the tendency in language design to improve the expressiveness of the language with new language constructs.

8.2 Summary

Languages are grown by adding new language constructs and we wish to preserve the decomposition into these constructs in their implementation. Language constructs intrinsically take into account other language constructs and therefore compromise their own modularization. Over the years, this problem has been the subject of much research in the domain of compiler technology (see Chapter 2). We have shown in Chapter 4 that, unfortunately, the mechanisms provided by contemporary language development technologies (introduced in Chapter 3) for separating a language implementation into modules do not suffice.

In this dissertation, we have defined a formal model (see Chapter 4) in which the grammar and the translational semantics of language constructs are modularized. In order to do so we impose three requirements on the translational semantics, one requirement on the grammar and one on the program representation. The requirements do not restrict the expressive power but, if adhered to, modularize a language implementation according to its language constructs. A result of those requirements is that we use a new definition of translational semantics in which we distinguish between two concepts: *definition* and *effect*. The *effect* of the translational semantics of a construct is the part

of its semantics which involves or interacts with other constructs. The requirements exile this part. The translational semantics of a construct that is not part of the effect merely *defines* its translational semantics.

From our model we have derived three kinds of language concerns: basic language concerns, language specification concerns and special-purpose language concerns. A basic concern consists of a *modular* language construct which is defined in *isolation* from the rest of a language implementation. It contains the construct's syntax specification and the definition part of the construct's translational semantics. Upon the introduction of a basic concern in a language, its stripped translational semantics is completed with its effect. The effect is expressed by establishing the necessary interactions between the language concerns of a language. The interactions use special-purpose concerns which describe reusable interaction mechanisms. For each of the three kinds of requirements restricting the definitions of translational semantics, there is a corresponding special-purpose concern offering a mechanism to complete it. We have determined the design challenges to implement each of the concerns in a modularized implementation module.

As an implementation of our modularization model, we have created a new open language development technique (see Chapters 5 and 6) through a tailored metaobject protocol (MOP) where languages are defined as a set of modular and separately defined constructs.

As a way to validate and prove that an implementation for such a MOP is feasible we have constructed a new language development technique called the Linglet Transformation System (LTS).

In the kernel of LTS (see Chapter 5), a language implementation is conceived as a set of interacting *linglets*, each capturing a basic language concern. Each linglet is responsible for representing a program fragment of a larger program and defines the operations on that program fragment. It effectively captures all the behavior of that language concern through compilation.

The interactions between linglets are implemented by using the three kinds of special-purpose concerns which we identified and explored based on our modularization model. These lead to the formulation of *interaction strategies* which define the mechanisms which can be used by linglets to establish their interactions. In order to construct interaction strategies, we have equipped the core of LTS with meta-facilities (see Chapter 6). We have chosen to open up LTS by means of a tailored linglet metaobject protocol called LMOP. In LMOP, interaction strategies are implemented as separate extensions of language implementations. LMOP allows us to use the basic semantics of the kernel of LTS and isolated linglets as such and customize their behavior in order to transcend their isolation and allow them to participate in complex interactions.

LMOP is orthogonal to LTS. This enables a form of unanticipated control over linglets that ensures the separation of interaction strategies and basic concerns, and ensures the necessary latitude to construct appropriate interaction strategies. In addition, the metaobject protocol supports the reuse of existing discretely defined interaction strategies through specialization and customization.

Complex translational semantics of a construct involving many interactions with other constructs, is highly cohesive and lowly coupled in a linglet, as the behavior of its produced individual program fragments can be controlled throughout the execution of

a language implementation by locally tweaking their behavior using LMOP. This fine granularity of LMOP is supported by the prototype-based object-oriented paradigm.

Each interaction strategy is captured in a discrete extension of LMOP in a separate concern. Interaction strategies can be shared and reused across different language implementations. More importantly, new interaction strategies can be defined, and existing ones can be specialized in order to meet and tweak interaction strategies for the separation of a concern challenge in a particular implementation. Furthermore, by taking into account a specific language implementation, language designers can exploit specific characteristics or structural properties of the language. Although this renders an interaction strategy dependent on a language, the result is an implementation with an optimal separation among the linglets implementing the various language constructs.

With this new and open language development technique, languages can be defined in terms of modularized language constructs while keeping the interactions among language constructs and interaction strategies implementing these interactions separate. We have validated our approach by developing a family of domain-specific languages using a shared pool of language constructs and interaction strategies (see Chapter 7), and by implementing the metalanguages we conceived in this dissertation (see Chapter 6).

8.2.1 Thesis

The goal of this dissertation is to design an approach for defining and constructing languages as a set of modular and separately defined constructs. It is our thesis that an open design of a new language development technique through a metaobject protocol is capable of modularizing languages according to their language constructs.

We have achieved that goal and validated our thesis:

- A language construct can be modularized by defining it in discrete language modules called linglets, isolated from the rest of a language.
- The interactions among language constructs due to their complex translational semantics are added in a language specification.
- Mechanisms for implementing the interactions among language modules can be defined separately as extensions of the linglet metaobject protocol.

In the remainder of this section we briefly summarise the main contributions of this dissertation.

8.2.2 Survey of Contemporary Language Development Systems

There are many different approaches, formalisms and techniques for implementing languages. Since we were pursuing modularization, we presented a detailed and extensive study of compilation based approaches in Chapter 3. This study is conducted by evaluating a number of common properties which impact modularization. We first analyze data structures for representing programs as they determine the possible operations used to transform and manipulate program representations. Subsequently we

determine how the set of operations offered by a system are used for implementing a transformation with local and global source and target scopes.

Each LDT provides insights in how to divide a language implementation and what mechanisms are necessary to maintain that separation. Our study discusses the strengths and weaknesses of each mechanism that individual language development techniques offer for modularizing language implementations. We conclude that transformation modules are implicitly co-operating and that it is often unclear how these relate to language constructs.

Our study also identifies a number of mechanisms that can be used to reduce the implicit coupling between modules. These successful mechanisms are categorized as interaction strategies for establishing co-operations between modules.

8.2.3 Modularized Language Construct Model

We have presented in Chapter 4 a formal model describing the modularization of a language into language constructs, which is formulated as a set of requirements on the translational semantics of a program.

The model modularizes language constructs that endanger their modularization because in general, they intrinsically take into account other language constructs. The modularization is defined for both their syntax definitions and their translational semantics because:

- The syntax of language constructs is described in terms of other language constructs. Hence, the syntax of language constructs is coupled with the syntax of other constructs.
- The translational semantics of language constructs cannot always be expressed with a simple homomorphic mapping. Instead, a more complex mapping must be used, in which language constructs are designed with other language constructs in mind. Hence, the translational semantics of language constructs is coupled with the semantics of other language constructs.

The model formulates five requirements for modularizing the syntax and the translational semantics of a language construct: one on the grammar of language constructs, one on the program representation and three on the translational semantics. The requirements ensure that the syntax and the translational semantics can strictly apply to a single language construct, even if it means that:

- The syntax of language constructs refers to other language constructs.
- The translational semantics does not yield a correct phrase in the target language.
- The translational semantics does not compose.
- The translational semantics of a language construct requires information external to the language construct.
- The translational semantics of a language construct exercises an effect on the translational semantics of other language constructs.

The modularization requirements have led to the definition of the following three kinds of concerns in a language implementation:

- A basic language concern representing a modularized language construct
- A language specification to complete the syntax and the semantics of modularized language constructs
- Three kinds of special purpose concerns corresponding to the three requirements on the translational semantics. These concerns respectively handle compositionality conflicts, multiple inputs and multiple results.

The evaluation of the suite of contemporary language development techniques (LDTs) against our formal model revealed that their language construct's modularization is inadequate. Furthermore, it shows that their interaction strategies for implementing special-purpose concerns are incomplete, fragmented and non-orthogonal to the LDTs. Given the current configuration of the interaction strategy space (see Section 4.5.3), LDTs do not offer sufficient means to separate the basic concerns. A closer analysis performed in Section 4.5.4 reveals four shortcomings of contemporary interaction strategies: they are not generally applicable, there is room for improvement, there is room for new interaction strategies and there is no silver bullet interaction strategy. We found that in order to be able to construct new and tailored interaction strategies, LDTs must be equipped with a generic meta-facility.

8.2.4 Kernel Transformation System

The Kernel Transformation System is the core of a new LDT we designed in Chapter 5 for solving the requirements imposed by the formal model on basic language concerns. Our implementation of the system is called LTS.

In LTS, languages are constructed from a set of isolated language modules which we call linglets. Each linglet implements a basic concern i.e. it defines the syntax of a language construct joined with the *definition* part of its translational semantics. Overall language semantics are defined by establishing interactions and co-operations among the linglets of a language.

Each linglet represents a language construct responsible for maintaining a program fragment and its manipulations. The language used to define linglets is a prototype based object-oriented language. As such, linglets and their instantiations, which represent particular program fragments, can be modified with adding behavior. It allows us to make linglets responsible for effecting their own semantics to other language constructs:

- A linglet defines the need for parts and is extensible, such that its parts can be bound to other linglets and that compositionality conflicts can be resolved.
- A linglet defines the need for external information and is extensible, such that this behavior can be provided in a language specification.
- A linglet defines the results its produces and is extensible, such that its results can be made responsible for integrating them into the target program.

The resulting high cohesion of a linglet and the low coupling among linglets clearly show that linglets are modularized and maintain strict separation.

8.2.5 Metafacilities for Defining Interaction Strategies

Basically a linglet can only access its parts. Access to other linglets was purposely kept so simple because the interactions and co-operations among linglets are so diverse that LTS must be designed to be extensible with interaction strategies to implement those interactions and co-operations.

Linglets co-operate and interact with one another using interaction strategies. Interaction strategies compensate for the primitive communication mechanisms of the kernel by offering new mechanisms for establishing collaborations among linglets that have implicit roles. Implicit roles enable linglets to interact without explicitly referring to others, using their relative location in the source or target program. As such, the co-operations are less dependent on other linglets and their composition. The reduction in dependencies improves the separation among linglets, even in case of complex co-operations and interactions.

In Chapter 6, we have defined a reflective layer for LTS via a specifically designed metaobject protocol called the Linglet MetaObject Protocol (LMOP). With LMOP, interaction strategies can be defined which can reflect about the structure and behavior of linglets. The former is needed to execute the logic of interaction strategies e.g. for the retrieval of information residing in other language linglets, or having changes to the results produced by other linglets. The latter is needed to trigger interaction strategies e.g. the provision or computation of information and the initiation of the integration and relocation of multiple results.

The reflective layer offers the following benefits:

- LMOP is a generic meta-facility useful for implementing a wide range of interaction strategies. As such, the most appropriate interaction strategy can be defined for the language implementation at hand.

- Linglets do not have to anticipate any interaction strategy by including additional responsibilities. The modularization of linglets is thus preserved.

- The co-operation among linglets can easily be lifted to the meta-level as the structure and behavior of the reflective layer mimics the concepts of the basic layer.

- The prototype-based model of the kernel is also present at the meta-level rendering interaction strategies highly extensible and reusable because they can be customized down to the level of an individual program fragment. Extensibility is an essential feature for some interaction strategies, as in the shift from base to meta-level a concrete language setting is lost. By extending the interaction strategies for particular languages, language-specific logic can be incorporated into these interaction strategies while keeping the generic part of the interaction strategy separate and thus reusable.

With LMOP, a compile-time MOP is defined allowing interaction strategies not only to change the run-time behavior of LTS but also the compile-time behavior. The compile-time MOP allows interaction strategies to offer syntactic convenience and static checks. This was accomplished by using the structural reflection of LMOP for implementing the two metalanguages which we conceived in this dissertation: a language for defining linglets and a language for defining specifications of languages. By changing those languages with strategy-specific extensions, special-purpose syntax for interaction strategies and static checks can be implemented.

8.2.6 New Interaction Strategies

LMOP is a generic meta-facility for implementing a wide range of interaction strategies. We have listed the identified interaction strategies embedded in other language development techniques and indicated how these interaction strategies can be implemented using LMOP. We conducted two experiments in which two interaction strategies were implemented (Chapter 6). These are defined as a family of interaction strategy extensions rather than as a single monolithic module. We concluded each interaction strategy with a couple of extensions.

In our first experiment we implemented an *extension* of an existing interaction strategy for retrieving context information. It shows in detail that existing interaction strategies can be implemented on top of LMOP and even extended beyond their original conception.

In our second experiment we implemented an *entirely new* interaction strategy for declaring and specifying scattering of code fragments called the Incremental Non-local Results (INR) strategy. It is a pioneering example where we take the ideas behind the successful techniques from compositional generators and introduce them in a transformational setting. Second, in contrast to the first interaction strategy which reasons about linglets, the INR strategy invasively changes the behavior of LTS. The INR strategy is also our prime example of the extensibility potential of interaction strategies. We indicated several extensions categorized into integration and relocation extensions. Although both interaction strategies improve separation of interactions between linglets, the INR is unique because we designed a novel interaction strategy which effectively modularizes the scattering of results in the target program.

8.3 Limitations and Future Work

During our research we encountered several opportunities for improving and complementing the approach we proposed. In this section, we indicate several directions for future research.

8.3.1 Sandbox Isolation Model

The methods defined in linglets may only access their parts and their semantically equivalent target program fragments, in order to preserve their isolation. Access to

parts is indispensable as the translational semantics adheres to the principle of compositionality. In essence, the translational semantics of a linglet is defined in terms of the translational semantics of its parts. However, methods defined in a language specification which are provided to linglets in order to complete them, need to communicate with other linglets for establishing the necessary interactions required to effect their complex translational semantics. In other words, a definition of a linglet consists of methods with different permissions for accessing and communicating with other linglets.

In the current implementation of LTS, access to other linglets needs to be controlled by explicit checks in the metaobject protocol and its extensions. These checks are thus scattered across the meta layer and pollute the definition of interaction strategies. What we require is a sandbox isolation model similar to a sandbox security model [GAS98]. The definition part of a linglet must only execute inside its sandbox. It can do anything within the boundaries of its sandbox, but it can't take any action outside them. By confining the sandbox to the parts of a linglet and their semantically equivalent target program fragments, we can ensure their isolation without scattering checks all over the meta layer.

There are a number of technologies such as aspect-oriented programming [KLM+97] or context-oriented programming [CH05] which could be used to extract this scattering and clean up the definition of the interaction strategies.

Aspect-oriented programming (AOP) is a programming technique for modularizing the implementation of crosscutting concerns. An AOP module is commonly referred to as an aspect and consists of two main parts: the aspect functionality and a pointcut that describes where it is applied.

Another approach is context-oriented programming. It provides a means for associating partial class and method definitions with layers. The layers can be activated and deactivated during the execution of a program. When a layer is activated, the partial definitions become part of the program until the layer is deactivated. As such, the behavior of a program can be modified according to the context of its use.

In order to tackle our problem with AOP, a pointcut needs to capture the method calls executed from within methods defined in a linglet and that return other linglets. This pointcut is not easy to express if we want to refrain from depending on syntactical conventions. Context-oriented programming seems a better fit, as the run-time LTS system could activate or deactivate the metaobject layers. Which technique is the most suitable remains to be seen.

8.3.2 Global Consistency Management

Proofs about program correctness are highly valued. Such proofs are also important in compilers. Since in LTS the semantics of a programming language is expressed by producing a semantically equivalent program, we want to ensure the correctness of the produced program.

The basic approach, which is traditionally used, is to type parse trees. The type-safety of specifications can only guarantee syntactical correctness i.e. the output parse trees correspond exactly to the grammar of the target language [vdBK02].

Syntactic correctness is only the first step, typically we also want to state properties

about the semantic correctness of programs. An example of such a next step is presented by Chiyan Chen et.al. in [CX03]. The types of a program cannot be reflected in the type of its parse tree representation. This is unsatisfactory as such representations make it impossible to capture in the type system of the meta language various invariants in a program transformation that are related to the types of programs.

In [Bri05], Brichau pushes this idea even further and introduces the notion of composition conflicts. In his program generation technique called Integrative Composable Generators (ICG) (see Section 3.6) these arise when program fragments get composed integratively. An integrative composition involves the invasive modification of produced programs. Because integrative composition breaks the encapsulation of produced programs, it is likely that certain integrations cause broken functionality. Therefore, he argues the need for an automatic conflict detection mechanism for the integrative composition of produced programs. As this mechanism covers the whole program being produced we refer to this as *global consistency management*. Brichau illustrates his concepts with an example where a particular integration of produced program fragments causes that both programs now have to deal with the same datastructure. One produced program fragment might expect a set datastructure, while another might expect an ordered list. Sets and ordered lists are semantically different datastructures because a set do not store duplicate values, while the ordered list does. He concludes that by detecting compositionality conflicts we can increase the correctness of generated programs.

The kernel of LTS does not enforce a particular kind of correctness. However, LTS does offer various opportunities to offer a basic degree of correctness. Each linglet can intercept changes to its parts and ensure local consistency (Section 5.6.4). Syntactical correctness could be implemented with an interaction strategy as an extension of LTS (Section 6.2). Compile-time checks about language specifications can be added by changing the metalanguages LL and LSL (Section 6.4.2). However, the degree of program correctness offered by ICG cannot be attained in LTS.

Combining both LTS and ICG and their conceptual models is not straightforward as they are quite different. ICG features a constraint system in which a set of different logic metaprograms produce a program, integrating the results of the different metaprograms. LTS features a prototype-based object-oriented system in which a set of interacting linglets and reflective extensions produce a program. Exactly how LTS and ICG can be combined on a conceptual and technical level remains to be determined.

8.3.3 Incremental Language Development

The modularization of language implementations into language constructs allows us to grow a language and its implementation by incrementally adding and changing language constructs. LTS directly reflects the incremental development process through its concepts and mechanisms. Iterative and incremental development of languages is targeted at language constructs and their semantics. These are the building blocks for language implementations in LTS, as languages are defined as the product of discrete language modules, each defining a single language construct. A single specification defines a language, facilitating a coherent and consistent co-operative behavior between

the language modules.

In Chapter 7, we have showcased the incremental development of a family of DSLs as a validation of our modularization model and technique. However, more research and empirical studies are necessary to map the subsequent steps in an incremental development process to actions and mechanisms in LTS that support these steps.

An interesting experiment in this context would be to replay the evolution of an existing language and investigate to what extent LTS can equip and aid language developers to perform the evolution. A good example language for this experiment would be Java. It is a popular language which has been subjected to significant evolution and evolution proposals, and we compare our results with earlier work on the modularization of Java Compilers performed by Ekman [Ekm06], van Wijck [VWKSB07] and Petersson [PR04]. Other interesting examples are the domain-specific languages or models which target evolving systems (see Section 1.1.2) (the so called E-type systems [Leh96]).

8.3.4 Application in other Language Development Techniques

In Chapter 3, we studied an extensive suite of contemporary language development techniques. Compared to the prototype-based object-oriented nature of LTS, there are LDTs which are based on quite different paradigms such as rewrite rules, graph rewrite rules and logic meta-programming.

In essence, we could apply our modularization model to these techniques and/or extend these techniques with a reflective layer. As the paradigm and the modularization of these systems may be totally different, we also expect the reflective layer to be quite different. An example illustrating how different a reflective layer can be, is the technique of programmable rewriting strategies of Stratego [Vis01a]. This particular mechanism offers first class control over the execution of rewrite rules and the creation and destruction of rules in the rule set. The vocabulary (see Section 3.1.4 for more details) of this mechanism is different from the vocabulary used in this dissertation.

It remains to be investigated how feasible and how adequate the result of such an application would be. This application presents an opportunity for an interesting comparison and will most certainly raise new issues that need to be dealt with. In co-operation with Kurtev, we already performed some preliminary research in [CK07], where we compaired QVT [Gro02b] and ATL [JK05] against our modularization model.

8.3.5 Interaction Strategy Library

Interaction strategies in LTS are not implemented as monolithic extensions, but rather as a family of extensions. In our experiments (Section 6.3) we described two interaction strategies using a set of extensions. Each experiment was open-ended as we concluded with a couple of extensions. Our prime example of the extensibility potential of interaction strategies was the Incremental Non-local Results (INR) strategy, where we indicated several opportunities for further extensions which we categorized into integration and relocation extensions.

During our experiments we only touched the surface of a true interaction strategy library. In order to construct such a library, we need to revise existing interaction strategies, classify them according to their intentions, and decompose them in smaller building blocks called μ-interaction strategies. These building blocks should be organized into a logical or naturally intuitive structure. The nature of this structure (hierarchical, iterative, etc.) is yet to be determined. Each μ-interaction strategy should also indicate its relationship to other μ-interaction strategies. Similar to the idea of a pattern language [AIS77], these relationships can be used as grammatical and semantic relationships in order to form an interaction strategy language.

The combination of these μ-interaction strategies would then lead to a particular interaction strategy. Currently specialization is the only primitive combinator of interaction strategy extensions. More complex combinators are necessary to hide implementation details and to prevent interference between extensions of a single interaction strategy and among several interaction strategies. An example of exposing such details is given in Section 6.3.2, where the combination of μ-interaction strategies required the developer to write some combination logic into the language specification.

The study of an interaction strategy language can help us to discover undesired interferences as a result of the cumulative effect of multiple interaction strategies. These can be incorporated into the interaction strategy language such that conflicting combinations of interaction strategies and μ-interaction strategies can be avoided. Such a conflict arises between attribute forwarding [WdMBK02] and attribute grammars [Knu68]. Forwarding, inheritance and synthesis are three possible directions to redirect a request for an attribute. Choosing the correct one or establishing an order among these directions is acknowledged by the authors to be one of the difficulties of the combination of forwarding with plain attribute grammars.

8.3.6 Modular Interpreters

Our modularization model is formulated using denotational semantics. Although we had to use a highly structured semantical domain to express translational semantics, the model largely applies for domains which are used in interpreters. The only requirement which causes a problem is the completability of semantic values.

Interpreters are used to prototype languages and to experiment with various language constructs. Compared to compilers, interpreters are considered a light weight approach for language development. The access of run-time information during execution of a program facilitates debugging, reflection, etc.

Interpreters construct a semantic value and continue the computation with this value. In other words, postfactum changes to that initial value which is performed in compilers, would invalidate all the computations that depend on that value. Hence, semantic values may not be changed after they have been computed and processed. The impact of non-changeable values on the model is fairly limited as the model is defined as a set of requirements. The impact on the implementation to adhere to that "new" model is the actual challenge.

As the modularization model also largely applies here, the implementation of the model in LTS is also to a large extent fit for implementing interpreters. The interac-

tion strategies presented in Chapter 6 that reason about the source program are also applicable in the context of interpreters. The remaining interaction strategies which change the target program cannot be applied in an interpreter. An example of such an interaction strategy is INR, which handles the integration of multiple results. This and other interaction strategies need to be replaced by a fitting interaction strategy for interpreters. As values may not be changed postfactum, all the necessary information to construct the correct value needs to be computed first. For this, a linglet should separate its "main" semantics from its "non-local" semantics. As such, the necessary "non-local" semantics can be retrieved and applied without having to compute its main semantics.

Clearly, the research presented in this dissertation largely applies to interpreters but the implementation challenges and the impact on LTS require more study.

8.3.7 Model-driven Development

Model-driven Development (MDD) [TB03] advocates software development via models. Models abstract from implementation details and allow programs to be specified at a higher level of abstraction. MDD is a promising player in todays software development practice and is quite similar to domain-specific language (DSLs).

MDD is an approach to software development where models are created before or instead of writing source code. These higher level models are meta-data definitions of the application that is to be built, consisting of concepts and relations at a higher abstraction level. The degree of abstraction is not further specified, and must be determined for the problem at hand. Models are described by a meta-model. Meta-models formalize the range of applications by defining the vocabulary and well-formedness rules. Meta-models are in turn described by metameta-models. Metameta-models describe the structure and semantics of metamodel specifications. Although very abstract, it is the most important level as it formalizes the means to construct other metamodels.

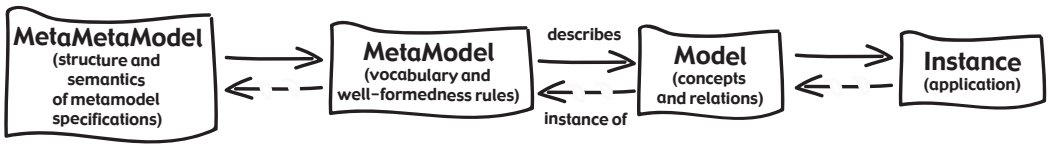

Figure 8.1: The four layered architecture of MDD

The MDD infrastructure [Gro02a] (see Figure 8.1) refines these higher level abstract models to lower level executable models, that is to say code. Exactly how this refinement from higher level to lower level models happens is not specified and is not standardized. Models get transformed into modules adding more technical detail until code is reached via code generators, transformation engines, template engines and model interpreters. A prime example of MDD is the Object Management Group's (OMG) Model Driven Architecture [Fra02, MM] (MDA) standard. MDA is an instantiation of the concept of

MDD by establishing a fixed set of metamodels (e.g. PIM, PSM) and by providing a specification for transformation engine to be used.

It is quite possible to take an iterative and incremental approach with MDD. This approach to MDD is called Agile MDD (AMDD) [Amb02]. With AMDD agile models are created which are sufficient for the iterative cycle at hand. Rational Unified Process [Kru00] (RUP), Enterprise Unified Process [ANV05] (EUP) and agile software development are capable of incrementally guiding the realization of a software application for which domains are neither completely specified and known upfront [Amb02].

MDD did not introduce a new language design technique or system but adopted and reused classical language implementation techniques. We investigated classical language implementation techniques in this dissertation and the specific MDD techniques in [CK07]. We can conclude that incremental and iterative design process used in MDD is not backed-up by an implementation technique that modularizes the MDD refinements in terms of language constructs i.e. modeling concepts defined in the metamodel. From a technical point of view it is feasible to apply our approach to MDD and we expect that MDD could benefit from the application of our approach.

The inverse is also true. It would be beneficial for our approach to be applied in a MDD context as MDD is complementary to DSLs: MDD is a graphical language and MDD places/integrates itself within the contemporary software development process. The graphical notation of models is designed for describing structures, while DSLs are better at describing business logic. The introduction into every day software development increases the need to evolve languages/modules as most of their applications are in a continuous state of flux [Leh96] and the application engineers are not professional language engineers. Hence, the application of our modularization model and reflective architecture in this context presents an opportunity for an interesting additional use of our approach.

8.3.8 Debugging

Languages are grown by adding more expressive language constructs, such that they can better reflect the intentions of the software developer. In fact, language constructs are used to hide implementation details from the developer. Unfortunately, in most language development techniques this hiding is abruptly punctured by errors raised by the implementation of the language. This is because they are formulated in terms of the implementation of the language. These errors are known to be cryptic because of the mismatch of abstraction levels. The end-user programmer has to understand the translated code in the target language, rather than the domain intention contained in the source language [Fai98].

We are currently facing the same problem in LTS. LTS does not offer dedicated features to hide the implementation of languages when errors should occur.

Furthermore, manual construction of debugging and error reporting for each new language can be time-consuming, expensive, and error-prone [WGRM05]. Therefore debugging support should be provided by the LDT.

LTS offers several advantages compared to other LDTs for adding debugging support. The first advantage is its modularization and its LMOP. In [WGRM05], Wu

describes an approach to add debugging to LDTs that reasons about the language implementation and that requires an explicit mapping between the source and target language. We believe that it is advantageous to apply Wu's technique in LTS as there is a clear correspondence between the properties of LTS and Wu's technique: In LTS the semantics of language constructs is modularized in linglets and LMOP reflects about these linglets. The second advantage LTS offers is the fact that LTS has been implemented in LTS. The meta-languages that define languages are also implemented in LTS. In other words, a language developer uses LTS itself to write languages. Hence, if generic debugging support is provided in the run-time system of LTS, not only the end-user languages would benefit from this feature but also the language developer.

In the future we plan to investigate the problem in more detail and reformulate it in terms of the run-time system. This would result in generic debugging support for language development.

8.3.9 Advanced Object-oriented concepts

Object orientation is acknowledged as a powerful paradigm to structure compilers. Attribute grammar systems like JastAddII [EH04], Eli [KW94], and Lisa [MuLA99] extensively make use of object-oriented modularization techniques. Of these systems, JastAddII is the most recent and advanced system. It uses two synergistic object-orientation mechanisms for supporting separation of concerns: inheritance for model modularization, aspects for crosscutting concerns.

The evaluation model of JastAddII is declarative and does not incorporate join points at the attribute level. Currently, only language constructs are present for refining an entire equation, similar to a piece of around advice, at a single join point.

It is useful to further investigate the use of point-cut languages of aspect orientation to refine interaction strategies. To the programmer, we should supply aspect-oriented (sub)languages that are based on the constructs and basic syntax that the programmer is most familiar with, as well as facilities for reifying and manipulating the crosscutting features of concerns. In this dissertation we continued pursuing the use of advanced concepts of object-orientation in language implementations. We have conducted research into a dynamic reflective object-oriented transformation language that provides a rich infrastructure for the development of aspect-oriented technology [Sul01a, Sul01b, BL02, Kic01, KLM+97].

Point-cut languages in LTS would enable language developers to easily and declaratively select linglets and linglet instances which need to be extended and customized with particular behavior. For example, a symbol table could be maintained by advising all linglets that have a syntactical parameter `name` with behavior to register themselves in a symbol table. Point-cut languages can also be used to shield the imperative and rather technical way of using interaction strategies via LMOP. Consider for example the Incremental Nonlocal Results (INR) strategy which we discussed in Section 6.3.3. The interaction strategy is customized for individual linglet instances in order to steer their integration in the target program fragments produced by other linglets. So, instead of overriding the methods `CORRESPONDS:` and `COMBINE:` of the INR strategy in their metaobjects, a point cut and advice could be used respectively. Our advanced experi-

ments with LMOP show the potential of LTS to incorporate new syntactic constructs together with their compile-time and run-time semantics.

8.3.10 Interaction Strategies for Aspects of the Semantic Behavior of Compilers

A lot of expertise in the community has build up to yield reusable and modularized language extensions, each providing additional functionality or an aspect of the semantic behavior of compilers. Research [Vis01b, Vis01a, vW03, OV05] in the context of Stratego is focused on the design of reusable interaction patterns to express rewrites. In [MuLA99] and in [OdMS00], templates and reusable attribute copy rules are specifically designed to share attribute communication patterns. During the design of ReRAGs [Ekm04] in JastAddII several useful transformational patterns were discovered, e.g. Semantic Specialization, Make Implicit Behavior Explicit, and Eliminate Shorthands. In addition, it has been observed that various language implementations also share a common design of name binding and type checking modules [Ekm06].

In LTS, we modularize the semantics of language constructs and modularize interaction strategies to establish the necessary interactions among language constructs. Compared to the literature, interaction strategies are currently generic and basic communication mechanisms like integrating nonlocal results, providing structure shy queries, etc. It would be interesting to evaluate various richer aspects of the semantic behavior of compilers in depth and combine efforts to design a shared library of them.

8.4 Perspectives

The central contribution of this thesis is all about the continuing effort to improve the process of custom-building new languages. We indeed witness an increasing need for this with the emergence of domain-specific languages and more recently with model-driven development. In this particular field of language research and engineering there is a significant gap left in how new languages and their meta-level structure co-evolve with their implementation. In this thesis we propose an initial approach to conceiving a language as the configuration of an interacting set of modular language constructs. Languages are developed in an open environment that is sufficiently flexible in tailoring specific design challenges. Language developers are at liberty to use custom language design constructs at generator-construction time and use static checking to improve the design process. At generator-execution time they can reuse, customize and construct interaction strategies to handle complex interactions among language constructs originally defined in isolation.

This thesis addresses new issues in language engineering using a novel approach. The first experiments proposed here may seem modest but they are significant and point the way to new and original principles of language design and implementation. Linglets are definitely in sync with the model approach to software engineering and contribute to the missing link between the abstract and the concrete view of building software.

Appendix A

Analysis of the degree of Separation of Concerns offered by Contemporary Language Development Techniques

In this appendix, we present the detailed analysis of the degree of separation of basic concerns and the three kinds of special-purpose concerns offered by contemporary LDTs.

In our discussion we revisit all of the language development techniques presented in Chapter 3. The references to the literature are not repeated here but can also be found in Chapter 3. Here is a list of the systems:

- Tree-based rewrite rules
  - Traversals (ASF+SDF traversals)
  - Dynamically scoped rewrite rules with rewrite strategies (focus on Stratego)
- Graph rewrite rules
  - Plain graph rewrite rules (like AGG)
  - Implicit Node Creation (like aspect-driven transformation systems)
  - Matching by Morphisms (like delta-rewriting and variable number of edges and nodes)
- Macros (focus on Macros designed for Common Lisp)
- Template-based Approaches (such as XSLT, Velocity)
- Attribute Grammars
  - Higher Order Attribute grammars
  - Forwarding
  - Reference attribute grammars

  - Multiple inheritance + Templates for common attribute communication patterns
  - First Class Attribute Grammars
- Compositional Generators
  - Subject oriented programming (SOP)
  - GenVoca
  - Integrative Composable Generators (ICG)
- Adhoc
  - Delegating Compiler Objects (DCO)
  - Intentional programming (IP)
  - Jakarta Tool Suite (JTS) in which we focus on the use of symbol tables and the use of general purpose language features
  - Functional programming languages in which we mainly focus on monads.

The structure of this section mimics the structure of Section 4.3. We start with discussing the basic concerns. The subsequent three sections treat the three kinds of special-purpose concerns.

A.1 Separation of Basic Concerns

Basic concerns describe a single language construct by defining its grammar and its translational semantics.

Only a couple of LDTs like DCOs, IP and Macros offer transformation modules which contain both syntax and semantics. Except for the above mentioned LDTs, all other LDTs separate the grammatical definition of the language constructs from their semantical definition.

A.1.1 R4 - Higher Order Grammars

DCOs offer a set of operators for reusing and extending entire grammars. In other words, the granularity of DCOs is too coarse and its grammars are merely chopped up into different modules: the productions still explicitly reference one another. The need for more flexibility is acknowledged by researchers working on JAMOOS and TaLe. In that work, the language constructs are separated into discrete modules. They are not successful because they lack a flexible binding technique in the grammar: “It was not possible to restructure the abstract grammar; a limitation which proved to be annoying in actual language definition.” [JY01]. So DCOs have an unacceptable degree of coupling between their basic concerns.

Only IP and Macros support a fine-grained granularity down to the level of a single language construct. In IP it is rather unclear how the grammar of the intentions is

exactly defined. As there is no real notion of a language in IP (a language is merely a set of intentions), we cannot also no access the degree of coupling between the intentions.

Macros are in that regard much mare tangible. They are compile-time rewrite rules using concrete syntax. Their syntactic definitions do not refer to other macros. Hence, macros separate the syntax of their language constructs. Unfortunately macros can be arbitrarily combined. In other words, macros define a trivial language and thus do not define a language as well. A macro could perform some tests on the bound code fragments to enforce the grammar of the language, but this would embed the grammar within the translational semantics of the macro. As our goal is to modularize the syntax, clearly this is unacceptable.

A.1.2 R0a - Partial Values Using the Bottom Value ($\perp$)

Only attribute grammars and template based approaches enforce that terms should be complete upon construction, and as such do not support a bottom value. LDTs which are embedded in the target language like macros, some compositional generators (like GenVoca and SOP) and template-based approaches entirely depend on the ability of the target language to represent incomplete program fragments. Likewise, these LDTs also depend on language features to support the completion of those fragments. In most languages this is very constrained. For example, the body of a method can be omitted in case its class and the method itself is declared abstract. For some cases ad-hoc solutions exist such as attaching an empty body to a method. However, the intention is not explicitly stated, which complicates the logic of further manipulations of the code (see the integration of multiple results). In both tree or graph rewrite rules, an additional symbol could be introduced which denotes the $\perp$ element. Unfortunately, the grammars used in tree based rewrite rules would have to be changed so that terms which contain a $\perp$ value remain syntactically correct. The change in the grammar is not only invasive, it also complicates the generation of a parser. Ad-hoc LDTs and the compositional generator ICG use general-purpose data structures to represent the target program. Most of these languages offer a $\perp$ element.

A.1.3 R0b - Completable Values

Completable values are important prerequisites for the definition of basic and special-purpose concerns. More precisely, basic concerns uses bottom values, and produces results that must be integrated in the values produced by other concerns. Completable values are supported by the bulk of LDTs, except for LDTs with implicit target structures like template-based approaches, GenVoca (see multiple results - identification in Section A.4.2 and A.4.1). An exception to the above is ICG. ICG offers an interface which allows program parts to be changed in a controlled fashion.

In addition, there is one other LDT which values are immutable, that is attribute grammars. By using higher order attribute grammars, multiple transformation stages can be concatenated, offering a way to alter values once they are produced. Staging transformations scatters the semantics of the basic language concerns and is thus also not an acceptable solution.

Note that although in functional approaches a value is not changeable, this can be solved with monads.

A.1.4 R0c - Semantics to Preserve the Local Consistency

As any language concern might change the result of any another language concern, a language concern should enforce the local consistency of its produced values. Contemporary LDTs, except the compositional generator ICG, do not offer support to ensure the local consistency. ICG takes an extreme approach in which changes to a value are strictly controlled by the transformation module. Only changes anticipated by the modules are allowed. This extreme view ensures complete consistency, at the cost of reusability and unanticipated change by other concerns for example for optimization, and integration of nonlocal results.

A.1.5 R1 - Compositionality

Every LDT adheres to the principle of compositionality i.e. the semantics of a program fragment is defined in terms of its parts, and exposes a composition rule which produces the target program fragment.

In some systems such as tree rewrite rules the composition rules might be a bit obfuscated but are never the less present. In tree rewrite rules a composition rule is manifested in the conditional clause of the rule. Requirement R1c which prohibits access to the nested parts of a program fragment is violated in all LTDs except for those in which the target program is implicit (see multiple results - identification in Section A.4.2 and A.4.1).

In all LDTs composition rules have access to the source language construct, and thus violate (R1a). In Section A.3 we discuss to what extent LDTs offer support to avoid this access.

Whether or not transformation modules need to cope with compositionality conflicts (requirement R1b) is discussed in Section A.2.

A.1.6 R2 - Multiple Inputs

Transformation modules of every contemporary LDT can consume extra information (R2a).

The need for extra information cannot always explicitly be stated by transformation modules. In the worst case, requirement R2d and R2b is violated and the input is implicitly collected upon consumption by the language concerns themselves. Template-based approaches and target-driven approaches in general fall in that category. The transformation modules (called templates) of XSLT for instance, embed direct XPath and/or XQuery rules. Note that newer versions of template-based approaches and target-driven approaches support parameterized transformation modules next to embedded queries.

Less severe are the LDTs which cannot distinguish between what information is locally available and what information is external to the module. Although there are clear advantages to be able to treat both kinds of information similar, totally lacking

the ability to distinguish at all complicates matters a lot. This is the case for the tree and graph rewrite rules and for compositional generators like GenVoca and SOP. These LDTs solely operate on their input, respectively the terms and the components to compose. Consequently, when more information is necessary, the input of the transformation modules is extended. In the extended input, the additional information is not distinguishable from the original inputs. This complicates the provision of inputs (see multiple inputs).

Other LDTs distinguish but nevertheless create implicit dependencies between the language concerns, the transformation modules and the overall language implementation. The transformation modules directly invoke an auxiliary function to retrieve the information. This function must be available in the implementation of the language for the language concern to work. Examples of such LDTs are rewrite rules with helper functions, conditional rewrite rules and attribute grammars. Rewrite rules with helper functions explicitly call auxiliary functions, conditional rewrite rules explicitly invoke other rewrite rules or traversals, and attribute grammars simply request the name of an attribute from either their parents or children. In all these LDTs the language concerns and the transformation modules must be able to respond to the call issued.

Attribute grammars are actually a bit more troublesome. Although they locally declare their attributes, they also request attributes from other language concerns. Therefore they cannot be considered to be specific for a language concern. We will see that they provide ways to partially reduce this problem (see identification and obtention of multiple inputs in Section A.3). Also information provision gets more complicated (see provision of multiple inputs in Section A.3.5).

In order to preserve the separation of basic concerns when consuming additional information, the need for information must by explicitly declared. As such, the basic concerns remain completely independent from the language implementation allowing concerns to be used in a changing language. Primary examples of such LDTs are the ad-hoc approaches and ICG which are based on general purpose languages. General purpose languages offer abstraction and modularization mechanisms to define interfaces for code. Using these interfaces, a module can indicate its need for extra information. ICG explicitly exposes empty program parts which need to be provided by other transformation module. However, the source program is directly queried from within other the transformation modules.

Whether essential additional information, required for other language constructs, can be produced (R2c) is discussed in Section A.1.7.

A.1.7 R3 - Production of Multiple Results

There are LDTs which prohibit the production of more than one result of a transformation module (R3a). Tree-based rewrite rules can only produce one single term which substitutes the matched term. Macros boil down to compile-time rewrite rules, which render them incapable as well. Template-languages and macros are embedded in the target language, so a transformation module cannot produce artificial structures containing multiple results such as cons-cells (see next).

Most LDTs do allow a transformation module to produce more than one result.

Unfortunately, the bulk of these systems do not offer explicit support or mechanisms to produce them. Consider the work-around to implement multiple results in rewrite rules [Cle05]. The idea is to produce an intermediate representation (for example with pairs, cons cells or extended terms) which combine local and nonlocal results. Intermediate structures complicate the resolution of the multiple results (see Section A.4) and scatter the semantics of the basic language concerns across several phases. Extensions of the rewrite rule formalism which support dynamically scoped rewrite rules, allow a rewrite rule to define new dynamically created rewrite rules. These rules can rewrite other terms to include an additional result. As such, multiple results can be produced. Ad-hoc approaches which are based on general purpose languages possess the ability to define them.

Graph rewrite rules, attribute grammars and ICG offer explicit support for multiple results. Graph rewrite rules can produce any number of results, but these results need to be attached to other nodes. As such, the production of results directly involves integration, breaking the separation between mutual basic concerns and between base and special-purpose concerns. Graph rewrite rules with implicit node creation do not need to attach all their results to the main graph on condition that they must be uniquely identifiable. If they are not identifiable, then there is no way of retrieving the results and integrating them. Hence, this interaction strategy is severely constrained and therefore not feasible. Graph rewrite rules use their flexible data structures to produce intermediate graphs. Compared to tree rewrite rules, the flexible data structures do no impose typing issues, but do not resolve the complexities due to rewriting.

Higher order attribute grammars can produce an intermediate version of the target program and subsequently produce the final program. Most attribute grammars choose to store their different results in different attributes. Anyhow, attribute grammars can produce multiple values while ignoring their further processing and thus respecting the separation of the base concerns.

ICG and DCOs produce larger program fragments. Unlike DCOs, in ICG transformation modules offer an integration interface which publishes the results produced by the transformation modules.

In LDTs where nonlocal results can be produced, nonlocals can also be consumed in order to express the semantics of a language construct (R3b). However, only essential nonlocal results may be consumed, and thus the challenge is to be able to handle these nonlocals separately from the transformation modules that produce them. Although, LDTs do not provide explicit support for this, work-arrounds are possible by phasing the transformations. We describe such a work-arround in [Cle05] for (graph) rewrite rules.

A.2 Separation of the Special-Purpose Concern - Compositionality

A.2.1 SP1 - Localized Interventions

Localized interventions are interesting because their effects are not visible to other concerns. Hence, such interventions preserve the separation among the different concerns of the language implementation.

In LDTs where the source program is mutable as in tree and graph rewrite rules, one way of implementing interventions is to change the source program. However, they obviously fail to localize their interventions. A rewrite rule can rewrite the source language program prior to applying the rewrite rules which encode the translational semantics. As rewrite rules are destructive, the changes in the source program are visible to all other transformations i.e. basic concerns, and other special-purpose concerns. Hence, these changes create dependencies. Graph rewrite rules can easily create new data structures. They could thus create a dual representation of the source program, to preserve the original graph. However, it is unclear how we can force a rewrite rule to operate on either of the two structures without hard-coding it into the rewrite rule.

In the remaining LDTs, where the source program is immutable, a localized intervention requires altering the behavior of the transformation modules with additional logic in order to cope with the conflicts. Except for some ad-hoc approaches (see later), there are no LDTs which currently offer an external mechanism for resolving compositionality conflicts. Transformation modules must either be designed to cope with a number of conflicts or must be altered when a language is to be constructed out of a given set of transformation modules. This approach is pushed to the extreme by ICG. ICG produces a larger program component out of smaller program parts. These parts are each implemented with a logic program which may yield different solutions. Each solution is an adaptation of the part to avoid a conflict.

DCOs are able to solve compositionality conflicts in an interesting way. As the grammar is chopped up into coarse grained fragments, the composition of these grammars is an opportunity for solving composition conflicts. A grammar production can be overridden by another grammar. Now, suppose the latter grammar is the composition grammar, a production can be altered to refer to a dummy production, which in turn refers to the actual non-terminal. In a later phase of the compilation process, these dummy productions serve as hooks to resolve composition conflicts.

Ad-hoc approaches have the potential to exploit general purpose language features (like higher order functions, inheritance, higher order logic) which are proven ways to influence the behavior of a module while respecting its modularization. Despite this potential no use of these features has been reported in ad-hoc approaches, except in functional languages and macros. Functional languages use monads to intervene in the composition of basic language concerns. More details are given in Section A.2.2. Higher order macros can produce other macros which are tailored to a specific context. Hence, resolution of compositionality conflicts can be provided. However, macros still have to be explicitly designed to be adapted with compositionality resolution logic.

A.2.2 SP2 - Global Interventions

Global interventions affecting the composition of multiple basic concerns are only possible in LDTs with mutable data structures. The most simple but rather crude approach is to change the source program before transforming it into another program, so one can globally intervene. Besides its crudeness, this approach is not always feasible because global interventions require much more subtle and more complicated logic. In other words, the logic requires details about the results produced by the transformation modules.

LDTs with mutable data structures therefore use intermediate versions of a transformed program in order to resolve composition problems after an initial transformation has taken place. Consider for instance the interaction strategy of producing multiple results in tree rewrite rules. In the first transformation phase, the results are contained in an intermediate representation, say cons cells. Resolving these compositionality problems is handled in successive rewriting phases. Having mutable data structures is not sufficient. Global interventions are not possible in ICG, SOP and ad-hoc approaches as one lacks the ability to post-process the intermediate structure within the boundaries of a LDT.

Global interventions can be externally defined with monads. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required [New]. As such, the recursive application of the compile function can be controlled by a monad. The translational semantics of language constructs can be made oblivious to the monads, hereby improving the separation of concerns (SOC). This kind of control is exactly what is required to implement global interventions.

A.3 Separation of the Special-Purpose concern - Multiple Inputs

A.3.1 SP3 - Identification with Abstract Names

In most transformation systems a transformation module captures a single task, i.e. the production of new terms. We have seen that external information is indispensable to perform that task. Controlling the information flow for a specific external information request by using a separate concern requires a way to identify the derived information: the requesting concern and the other concerns involved to compute the information.

Identifying the need for external information is only possible in cases where the consumption of external information is explicitly declared (see basic concern - multiple inputs in Section A.1.6). Tree rewrite rules and macros plainly state the required information as parts, with no distinction between what is external and what is local. The requested information is a derived value. One of the challenges in this task of the concern is to keep the logic to compute the derived value which is specific to a concern local to that concern. This logic thus needs an abstract name, so as to be able to use the information and provide it to the requesting concern. The issue of keeping the

specific logic local and the provision of the information is discussed in multiple inputs - provision of information in Section A.3.5.

Abstractly identifying the derived information is supported by adhoc approaches, graph rewrite rules, rewrite rules, macros, ICG and attribute grammars. Adhoc approaches use general purpose language constructs. Graph rewrite rules can easily create new nodes and edges. Hence, they can label nodes with an abstract name in order to obtain the value. Rewrite rules equipped with helper functions, can call a helper function. Rewrite rules with traversals could misuse an accumulating traversal that does not traverse, but computes an additional value. The integration interface of transformation modules in ICG provide abstract names for its program parts. They can be accessed as such.

Attribute grammars and ad-hoc approaches that employ a symbol table offer the best support, because the names are location independent and thus completely preserve the isolation of the concern even during communication. Attributes are known to every term. Attribute computations use attribute names to retrieve attribute values and attribute computations use names to assign the computed the value of that attribute. Identifying and using the value is thus location independent. Symbol tables are very effective as the requestor of a value simply accesses the symbol table and retrieves a value by means of an agreed upon key. The key abstracts the concrete location and whereabouts of the information.

Oddly, template-based approaches do not support identification via an abstract name. A template transformation module can only produce target program fragments, not derived information from an arbitrary information domain.

A.3.2 SP4 - Obtention of External Information

External information is one of the most simple forms of information to locate. The information is external to the input program and must be made globally accessible. Globally accessing information was the very first thing programming languages could do. So, for ad-hoc approaches or other LDT which are closely related with a general purpose language such as compositional LDTs or macros accessing global information does not pose a challenge.

LDTs especially designed to implement a language according to a particular paradigm have been striving to keep their transformation module as modular as possible (cfr. basic concern - consume multiple inputs in Section A.1.6 and produce multiple results in Section A.1.7). As a result, accessing global information has been limited. The prime examples are rewrite rules and attribute grammars, which can only consume values which are part of the input program. The ease of accessing this information depends on query facilities and identification facilities. Rewrite rule systems need to traverse the input program and provide the information where necessary by rewriting the source program. Rewriting of the source program tree is fragile in case of multiple information requests (see multiple inputs - obtention of information of an another language concern in Section A.3.3). Conceptually global information poses no problem to attribute grammars as they retrieve location independent information. However, the attribute copy rule still has to be provided either manually or by the system. Template-based

languages, for example, are known for their powerful query facilities. Retrieving a value from anywhere in the source program is thus not a problem.

A.3.3 SP5 - Obtention of Information of Another Language Concern

In order to obtain information located in another language concern, the input program needs to be traversed and the derived value computed. Macros and rewrite rules can only access the matched subtree. Equipped with traversals, rewrite rules can describe complicated traversals. Equipped with rewrite strategies, several strategies and rewrites can be combined to describe descending paths. However, so as to gain access to information contained in the parent, a five step process needs to be followed:

- Determine a source program term that is a common ancestor of the matched node (pivot) and the terms that contain the information. This term serves as a starting point for subsequent operations.
- Find all the terms starting from the common ancestor.
- Collect the information contained in the terms.
- Propagate the collected information to the pivot node.
- Rewrite the pivot node to provide the collected information

Tree rewrite rules are destructive. The matched term is replaced by another term. Because terms can be removed, basic concerns that depend on that term to obtain information must first be executed. Moreover, multiple information requests get implicitly scheduled by the signature of the terms. This scheduling is an additional complexity which erects dependencies that cripple the separation of base and special-purpose concerns.

In macros, a similar process has to be used. There is one way to get the whole program reified that is defining a top-level macro that behaves like the identity function, something like (with-top-macro ... the whole program ...).

First class attribute grammars or attribute grammars with attribute templates can externally control the flow of information from one concern to another. There is however, a very limited form of control over the direction of information flow. Basic control can be exercised by either accessing the attribute values of the parent or one of the children. Even with templates or first class attribute grammars, the direction of a flow needs to be manually enforced by formulating attribute computations for specific productions. Furthermore, complex paths for retrieving information of another concern need to be specified by a set of attribute definitions scattered in several concerns.

The values of the keys in the symbol table must first be computed before the values can be accessed. Unfortunately, the maintenance of the symbol table (e.g. storing and removing the values) is embedded in the translational semantics of the basic concerns that produce the value. In some cases it can be argued that the manipulation of the symbol table can be considered as part of translational semantics (cfr. requirement

R2c). However, arbitrary information that is requested by a concern should in general not enforce additional behavior in the concerns that own the requested information.

General-purpose languages typically provide good implementations to query tree structured data if a suitable representation is used. Some require a meta-structure (e.g. container - element structure) to facilitate queries like traversals in OO, in Prolog and in functional languages like Haskell. Some require developers to use these metastructures, some generate them and others use reflection.

Template-based approaches and graph rewrite rules are by far the best equipped LDTs for retrieving information. Plain graph rewrite rules have a limited scope, but as arbitrary information can be added in the graph, information retrieval is often defined as a recursive process where each rewrite rule builds upon the results of another. Such implementations can rapidly become quite complex in case a well-defined path must be followed in the graph. Matching with general morphisms is a solution, but is difficult to manage. XML-based LDT like XSLT offer powerful query mechanism like XPaths or XQuery that allow precise paths to be described, but without requiring us to detail the concrete path that needs to be followed.

A.3.4 SP6 - Obtention of Distributed Information among Several Concerns

One of the characteristics of a concern is its high cohesion. All semantics that is directly related to that concern should therefore also be part of that concern. In the retrieval of distributed information, every concern involved performs a part of the computation which is specific for that concern. These computations should logically become part of those concerns. Hence, concerns should be modifiable or extendable with additional logic once we use a concern in a particular language.

Of the LDTs like DCOs, IP and macros that offer transformation modules which define an entire basic concern i.e. its grammar and its semantics, none have the ability to externally alter the modules with new behavior.

The best equipped LDTs are attribute grammars. New attribute definitions can be stated separately for each nonterminal which can override or complement existing definitions.

The remaining LDTs also perform rather poorly, only considering the translational semantics of a transformation module. In tree and graph rewrite rules, a number of successive rewrites must be used to traverse the input tree and make the necessary computations. As we said earlier, traversals improve the elegance but remain limited because they cannot access context terms. Template-based LDTs call different templates to act upon a term. The additional rules or templates are totally unrelated to the basic concerns. Except for ad-hoc approaches, all other systems simply call auxiliary module to perform the computation, and thus fail to be extensible.

Ad-hoc approaches have the potential to extend modules, due to the extension and parameterization mechanisms of general purpose languages: higher order functions, inheritance, etc. A clear need for these mechanisms is present in the symbol table strategy which is favored by some ad-hoc approaches. As the maintenance of the symbol table (e.g. storing and removing the values) is embedded in the translational

semantics of other basic concerns, all basic concerns involved must be altered with this responsibility.

A.3.5 SP7 - Provision of Information

The mechanism to provide information depends on whether the request for additional information is implicit or explicit.

Providing information by a special-purpose concern to a base-concern is not trivial. An important issue here is scheduling, i.e. making sure that the value is available when required. There are several ways of providing information, but all are based on the flexibility of a tranformation module or its data structures.

First, the base language concern is a flexible data structure which can be changed. There are not many LDTs which support flexible data structures. The only one is graph rewrite rules. Unfortunately, additional information is not explicitly stated, which render graph rewrite rules less useful for these purposes. Rewrite rules are excluded because in order to add information to a term in rewrite rule systems, the signature of the term needs to be changed. Rewriting the source tree introduces implicit scheduling between multiple information requests.

Second, the base language concern is extensible with additional semantics. In the latter case, the additional semantics can trigger the special-purpose concern to obtain the information. So there are less scheduling problems with the second solution. The only LDTs that qualify are attribute grammars and ad-hoc approaces. We refer to the previous section about distributed information in which this form of extensibility is detailed.

Third, a special-purpose concern can respond to a need and create an information flow when the information is requested. There is only a single LDT in which this is truly the case and to a certain extent the family of LDTs to which the LDT belongs. The LDT we are talking about is an attribute grammar supporting forwarding. Forwarding redirects requests for undefined attributes to its produced translational semantic value. In plain attribute grammars attribute copy rules are implicitly provided by the system. These rules redirect requests for undefined attributes to either the parent or one of its children. Extensions of attribute grammars provided explicit copy rules, but hereby lost the ability for an external concern to act upon an attribute request.

A.4 Separation of the Special-Purpose concern - Multiple Results

A.4.1 SP8 - Identification via the Source Language Program

Identification steered by the source program requires modifications to the producing source language concerns to enable changes to their produced results. The basic requirement is accessing the basic concerns and these basic concerns provide read-write access to their results. Higher order attribute grammars are ruled out because they

produce immutable values. Except for ICG, most of the LDTs with implicit target programs do not fulfill this minimum requirement.

Having an explicit target program is not sufficient either. Due to scheduling issues, transformation modules may or may not have already been triggered. This has several implications. First, the source structure must be preserved during the execution of the transformation. Tree-based rewrite rules and macros do not ensure this. Second, changes due to integration must be reflected in the target program, the concerns must yield the same value representing the target program fragment when called. Except ICG and graph rewrite rules, no LDT has been encountered that supports this. ICG explicitly support this through its integration interface. In graph rewrite systems equipped with implicit node creation, a similar effect could be implemented by explicitly linking the produced target nodes to the source nodes upon rule execution.

ICG features a static identification, which in cases of multiple integration locations causes a lot of scattering because it is statically determined where nonlocal results should be integrated.

A.4.2 SP9 - Identification via the Target Language Program

Embedded LDTs like template-based approaches and other LDTs like ICG, GenVoca which use an implicit target language program do not expose the target language program constructed while the transformation is executed. Hence, it is not possible to use the target language program to identify the term where a nonlocal result should be integrated. In macros, it is close to impossible, as in macros only the parameters are deconstructed. The rest of the program cannot be traversed.

SOP offers identification rules to define which program fragments need to be combined. The interaction strategy used to combine the fragments is separated from the identification rules. The rules can relate program fragments by uniquely identifying the program parts. The kind of identifiable program parts depend on the target language concepts down to the level of individual methods.

The remaining LDTs with an explicit target language program expose the data structures which can be accessed by other language concerns. Identifying a term in a target program with adhoc approaches, tree or graph rewrite rules is (technically speaking) rather simple, as the mechanism to obtain additional information from the source program can also be used to identify target program terms (see multiple inputs - obtention of information in Section A.3). What complicates matters is scheduling. The resolution of the nonlocal results must be scheduled carefully to ensure that the terms in which nonlocals are to be integrated, exist. The result is scattered integration logic over multiple rules and stages in the transformation process [Cle05].

Rewrite rule LDTs extended with dynamic rewrite rules produce a rule which contains the identification of the results. Although the mechanism to deal with nonlocal results is a separate mechanism implemented by the LDT, the identification is a fixed part of the rule. There are two ways of dealing with this issue: lightweight rules, or heavyweight rules. Both have their limitations. When opting for lightweight rules, rewrite strategies are necessary to control the application of the rules. Unfortunately

rewriting strategies also statically determine the terms in which these ought to be applied. In cases of multiple potential integration locations, statically deciding where the results should be integrated causes a lot of scattering. When opting for heavyweight rules, the identification cannot be altered externally if the language implementation changes. Moreover, the identification is constrained because the context of a term cannot be accessed.

Dynamic rewrite rules advocate an asymmetric approach. There are rules that create a term and there are rule that modify (rewrite) the term. However, it is in general undecidable which rule will be fired and thus undecidable which rule must be made responsible for the production of a target program fragment. In order to solve this issue, the notion of aspect-driven transformations was introduced in graph rewrite rules to obtain a more symmetric model. Aspect-driven transformations rely on implicit node creation. Implicit node creation in graph rewrite rules identifies the terms which need to be combined based on the ability to uniquely identify concepts. This ability depends on identification mechanisms of the target program language at hand and on the requirement that translational semantics always produce uniquely identifiable entities.

Higher order attribute grammars can produce an intermediate program of an intermediate attribute grammar. The latter contains the semantics for retrieving the nonlocals and integrate them. The identification is statically determined and tangled in the language concerns which produce the terms in which nonlocals must be integrated. As we said earlier, in case of multiple integration locations, statically deciding where the results should be integrated causes scattering.

A.4.3 SP10 - Scheduling

The complexity of scheduling largely depends on the ability to change produced program fragments on which previous computations (execution of special-purpose concerns or basic concerns) depend. If such is the case, proper scheduling must ensure that either no changes can be made to terms on which previous computation depend or that changes cannot invalidate previous computations. One of the two most powerful systems in this respect are ICG and rewrite rule strategies. ICG computes the correct integration of program fragments using a constraint network based on a manual composition. Alternative solutions are selected and propagated in the network when integration should fail. Rewrite rule strategies offer powerful rewrite strategies and traversals to manually specify the desired schedule of execution.

LDTs which have immutable target language programs are confronted with far less complex scheduling. Attribute grammars is one of the prime examples. Scheduling has always been considered a non-declarative issue, therefore attribute grammar systems must deduce a suitable scheduling. Immutability does not solve all scheduling issues. A prime issue is ensuring that upon identification, the term to find must already be produced. In systems were the separation between special-purpose and base-concerns is not properly kept, this is an issue that needs to be tackled by the base concerns. Such systems are attribute grammars. Another issue arises when nonlocals are nested. In this case, a breadth-first or depth-first strategy to resolve the nonlocals may easily

yield different results. Rewrite strategies are the only mechanisms in which this issue can be properly addressed as a correct scheduling external to language concerns can be defined.

Other systems must hardcode scheduling logic into the identification part of the transformation module.

A.4.4 SP11 - Integration - a Three-party Contract

Ideally, integration should be a three-party contract between the concerns that produce the nonlocal results, the concerns that produce the terms in which the nonlocals ought to be integrated, and an integrator dictating the composition. There are no LDTs in which such a contract can be implemented.

In LDTs with implicit target programs or immutable target programs, the base language concerns must be altered to handle the nonlocal results. This is the case or ICG and attribute grammars. In ICG, a base language concern allows controlled access to the produced program in order to ensure that integration is computed correctly. Attribute grammars use attributes that retrieve nonlocal results. In macros, GenVoca and template-based approaches no multiple results can be produced. In these cases, the production of the additional results becomes the responsibility of the concerns which produce the terms in which the nonlocals must be integrated. Clearly, the separation between basic concerns is violated.

ICG and SOP do offer external composition rules which serve as external actors that direct the integration. ICG shifts more control to the basic concerns. As a result the semantics of the integration rules of ICG is defined in the basic concerns, whereas the integration rules of SOP have their own semantics.

Integration with rewrite rule systems starts from an intermediate representation (for example with cons cells or extended terms) which combine local and nonlocal results. This intermediate representation is afterwards rewritten until the nonlocal results are correctly integrated in the target program. Rewriting this intermediate tree is a complicated process [Cle05]. In type safe tree-based rewrite rules should be altered such that the cons cells or extended terms are valid. The major drawback remains the fragility of the strategy: controlling successive rewriting is complicated and the semantics is based on naming conventions. Rewrite rule LDTs extended with dynamic rewrite rules produce a rule which contains the integration of the results. There is no way of excluding the integration of the term from the nonlocal result. So nonlocal results are packaged in heavyweight rules. As a result, integration is dictated and entirely controlled by the nonlocal result.

Integration of nonlocal results is either embedded in graph rewrite rules (see basic concerns - production of multiple results), or a similar strategy such as tree-based rewrites is followed (see above). Implicit node creation in graph rewrite rules integrates the terms which need to be combined based on the ability to uniquely identify concepts. The only actual integration operator is a union: different target program fragments are combined by merging the program fragments. Duplicate subfragments must be explicitly anticipated by the designer of the rules.

Higher order attribute grammars can produce an intermediate program of an inter-

mediate attribute grammar. The latter contains the semantics to retrieve the nonlocals and integrate them. The integration is statically determined and tangled in the language concerns which produce the terms in which nonlocal must be integrated.

Integration of nonlocal results can be implemented with monads. As monads control the recursive application of the compile function, the produced results can be integrated with nonlocal results after each invocation. The translational semantics of language constructs can be oblivious to the monads, thereby improving the SOC. For each integration a specialized monad needs to be written that ensures the consistency and the semantics of both the nonlocal result and the term in which to integrate the nonlocal. As such, these specialized monads break the separation of concerns of the involved basic concerns as they need to be aware of the details of the translational semantics of those concerns. The same conclusion was formulated by Brichau [Bri05] "the challenge remains in identifying the possible locations in a program that permit integration and prevent the breaking of existing functionalities in each transformation."

DCO acknowledges the need to resolve four kinds of interferences, but does not further specify how to deal with them.

A.4.5 SP12 - Integration - Context-dependent Integration

Besides the three concerns of the thee party contract, context-dependent integration involves an arbitrary number of other concerns which influence the integration semantics. So the integration logic must already be partially active during the identification step.

No LDT actually supports context-dependent integration. Due to the variable number of concerns it is basically tackled with the same mechanisms to retrieve distributed information. The major implication of this decision is the scattering of integration logic. Consider for example graph or tree based rewrite rules, these integrations are tackled by a traversal, traversing either the source program or the target program to extract the necessary information in order to compute the correct integration semantics.

Attribute grammars are by far best equipped with respect to other LDTs. Their ability to compute distributed information can be applied to compute the context-dependent integration over the target tree. Unfortunately, the integration itself is not supported and remains scattered over the target language concerns.

Appendix B

Analysis of Interaction Strategy Applicability

In this appendix we present a detailed analysis of the applicability of the interaction strategies found in contemporary LDTs.

We compare the interaction strategies that improve the separation between interacting basic concerns by focussing on the same task or challenge of the same special-purpose concern. There are four such tasks or challenges for which there is more than one interaction strategy (see Table 4.4):

1. obtention of information of the special-purpose concern multiple inputs
2. identification of information of the special-purpose concern multiple inputs
3. identification of the location of nonlocal results of the special-purpose concern multiple results
4. integration of nonlocal results of the special-purpose concern multiple results.

Needless to say that the task or challenge which only has one interaction strategy is also not generally applicable. The tradeoffs of these interaction strategies can be found in the discussion in Section 4.5.1. In the remainder of this section we show that interaction strategies implementing the same task or challenge of a special-purpose concerns have different tradeoffs and are not simply interchangeable.

1. *Obtention of information of the special-purpose concern multiple inputs by traversals, morphisms, structure-shy queries, attribute propagation rules and monads.* In order to illustrate the decision process, let us evaluate which interaction strategy is best suited to obtain additional information.

   Retrieving information from arbitrary locations is best supported by structure-shy queries. Such complex queries can easily be expressed locally in a single definition without needing to change the other concerns. On the downside, structure-shy queries still explicitly reference other concerns.

   Retrieving information encoded in a complex pattern between source and/or target language nodes is best expressed with graph rewrite rules using morphisms.

The left-hand side of a graph rewrite rule can contain an arbitrary number of nodes and interconnecting edges. The complexity of computing a morphism ranges from simple subgraph matching to more complex delta rewrite rules.

Retrieving distributed information among several concerns is best expressed with attributes in conjunction with attribute copy rules. In attribute grammars, one simply requests an attribute value instead of having to define which concerns define the attribute. By using copy rules, the concern which defined the attribute is reached and in turn, the necessary computation involving information of other concerns is triggered. The distributed nature of attribute computations can also be its major drawback as it scatters the logic to compute a value and the path to reach the value in case of complex queries.

Retrieving information which involves the entire hierarchical structure of a term in a controlled fashion is best tackled by traversals. Traversals locally define the actions that need to be applied to each encountered subterm, and declaratively specify the properties of a traversal i.e. the order, when the recursion should be continued or broken off, and the direction. However, the acquisition of information along a subtree does not suffice to retrieve information elegantly from arbitrary places in the tree, as traversals can only descend.

Sharing information along the execution trail of basic concerns is best achieved with monads. For example, the state monad propagates state information from one function application to the next one. This way, the translational semantics of language constructs can be oblivious of state, thereby improving the separation of basic concerns. Information flows that go against the execution trail are not handled elegantly in a monad.

2. *Identification of information of the special-purpose concern multiple inputs by symbol tables, attributes and attribute forwarding.* In order to illustrate the decision process, let us evaluate which interaction strategy is best suited to identify additional information.

   Identifying information that depends on the context of a language construct is best expressed with attributes or with symbol tables. There is in essence no difference between a symbol table and an attribute, except in their computation. Symbol tables are always computed whereas attributes are computed upon request. Attributes and symbol tables are global defined, but their value(s) are computed in a local context.

   Identifying information that depends on the program fragments being the translational semantics of language constructs is best expressed with attribute forwarding. Requests for attributes which are unknown to a basic language concern are redirected to the values produced by its translational semantics. As such, a basic concern "inherits" these attributes without having to redefine them.

3. *Identification of the location of nonlocal results of the special-purpose concern multiple results by dynamic rewrite rules, morphisms, implicit node creation, identification rules of SOP and ICG.* In order to illustrate the decision process,

let us evaluate which interaction strategy is best suited to identify the location where nonlocal results must be integrated.

Identifying the location of nonlocal results, in case of multiple integration sites, is best expressed by dynamic rewrite rules. By encoding nonlocal results into dynamic rewrite rules, the nonlocals can be integrated each time the left hand-side of the rewrite rule matches. Identifying the location of nonlocal results within a complex set of relationships is best expressed with morphisms. We use the same reasoning as in the obtention of information of the special-purpose concern multiple inputs.

Identifying the location of nonlocal results with unique identifiers is best expressed by implicit node creation. Implicit node creation identifies the terms which need to be combined based on the ability to identify concepts uniquely. This ability depends on the identification mechanisms of the target program language at hand and on the requirement that translational semantics always produce uniquely identifiable entities.

Identifying the location of nonlocal results in a fixed and upfront known set of generators is best expressed by compositional systems. SOP and ICG offer rules to establish whether two fragments correspond. SOP rules can establish correspondences between fixed number of uniquely identifiable program parts. Whereas in ICG arbitrary program parts can be exposed or hidden to deal with fine-grained and flexible compositions. However, due to the unrestricted access to the program parts, SOP is able to shift the composition logic out of the consuming concern into the composition rule.

4. *Integration of the location of nonlocal results of the special-purpose concern multiple results by implicit node creation, ICG, monads, SOP and dynamic rewrite rules.* In order to illustrate the decision process, let us evaluate which interaction strategy is best suited to identify the location where nonlocal results must be integrated.

   Interaction strategies differ in the weight of the integration which respectively lies in the concerns that produce nonlocals, in the concerns that consume nonlocals, in the external actor, in separate composition rules ensuring the target language semantics and (again) in concerns that produce nonlocals. As you can see, none of them actually support the three-party contract, but each of the interaction strategies achieves a certain degree of separation.

   Interaction strategies also differ in their support to integrate two target program fragments. Implicit node creation poorly supports the integration of two target program fragments. The only actual integration operator is a union. Composition rules provide a rich set of semantics for combining the program fragments. In ICG, the rules must be explicitly defined by the developer. In SOP a set of generic operators are defined which ensure the target language semantics. A suitable operator must be chosen for the fragment at hand. Monads are a vehicle which can be used to express the integration of nonlocal results, but do not assist nor provide specific mechanisms to perform the integration (see Section A.4).

Dynamic rewrite rules are also merely a vehicle. We therefore do not consider the later two as a special case in the decision process.

Interaction strategies also differ in their assumptions concerning scheduling. Composition rules are applied in compositional generators with a fixed and upfront known set of target program fragments. That is the case in ICG, SOP and dynamic rewrite rules. The generators are fixed and known upfront, and so are their target program fragments. Implicit node creation is an interaction strategy which is constructed for exactly the opposite scenario. In graph rewrite rules, it is in general undecidable which rule will be fired and thus undecidable which rule must be made responsible for the production of a target program fragment. The interaction strategy composes behind the scenes what should be composed.

Appendix C

KALA Language Specification

KALA [Fab05] is the language we use to define advanced transaction specifications. KALA was specifically created to allow for the separate specification of transactional properties of Java methods. In KALA, the programmer declaratively states the transactional properties of a Java method in one block of statements.

We first introduce the main language constructs in KALA. The formalism we use is LTS. So each of the language constructs is defined in a single linglet. The language specification of KALA is presented in Section C.2, combining all the introduced linglets together into a language. We omitted their translational semantics because the semantics of KALA is beyond the scope of this dissertation. We refer the reader to [Fab05] for a more in depth discussion regarding the formal background of the language and its translational semantics to Java. We conclude this chapter by listing some example KALA specifications for ATMS.

C.1 Language Constructs

Most language constructs are derived from its formal model ACTA. There are also additional constructs present in the KALA language supporting secondary transactions, termination of transactions and groups of transactions.

C.1.1 Transaction

A transaction declaration consists of a name and a series of transactional properties. The `Tx_Declaration` linglet, shown below, defines this declaration.

```
Linglet Tx_Declaration {
   syntax {  signature   "{" body "}" }
}
```

Transactions coincide with methods in the application. Hence, the signature of a transaction is the signature of a method. Transactional properties are associated with

the significant events in the lifecycle of transactions. The typical events are begin, commit and abort.

Transactional properties establish relationships between two ore more transactions. As such, transactions need to refer to other transactions, by using a public name service.

C.1.2 Naming

KALA programs can declare transactional properties of a method separately from that method's definition. Therefore, a static naming scheme is required to identify for which method a block of properties is intended.

Statically, KALA identifies a transaction via a method signature. In order to name the method, the full class name and the method signature, separated by a dot, are given. This is defined by the `Signature` linglet, which is shown below.

```
Linglet Signature {
   syntax {  (package ".")* class "." method
             "(" !(parameter ("," parameter)* ) ")"  }
}
```

To set properties on transactions at run-time, KALA offers dynamic naming. To obtain a reference to a registered transaction at run-time, KALA code uses the naming service. The transactions are registered with keys which are computed at run-time.

Lookup is performed in KALA by using an `alias` statement (see below), which takes as first argument the variable serving as the name of the transaction, and as second argument the key for the lookup operation. The latter can be computed by a Java expression. Within this expression, existing aliases can be used, along with the keyword `self`, which stands for the transaction to which this declaration applies at run-time.

```
Linglet Alias {
   syntax {  "alias"  "(" name "," key ")" }
}
```

Registration of transactions with names is performed by the `name` statement, with as first argument a variable containing the transaction to be registered and second a key. The transaction to be registered, is either the result obtained through an `alias` statement, or the `self` pseudo variable. Registering a transaction overwrites the previously held binding.

```
Linglet Name {
   syntax {  "name"  "(" name "," key ")" }
}
```

C.1.3 Grouping

Some transactional properties affect a group of transactions. The population of groups may depend on application logic, and is therefore, in general, impossible to determine statically. Hence, two additional statements are provided to subscribe and unsubscribe transactions in a group.

A transaction can be subscribed to a group with the `groupAdd` statement. The statement takes the same arguments as an `name` statement: the first argument is a transaction to be registered, the second is a key of the group. The group key is computed by a Java expression. References to groups are obtained though the `groupAlias` statement, which is analogous to the `alias` statement above, but now looks groups that have been previously registered using the `groupName` statement.

```
Linglet GroupAlias {
   syntax {  "groupAlias"  "(" name "," key ")" }
}
Linglet GroupAdd {
   syntax {  "groupName"  "(" name "," key ")" }
}
```

C.1.4 Significant Events

In KALA, the programmer declaratively states the transactional properties of a Java method in a block of statements, using the constructs provided by the ACTA formal model. In other words, in KALA, dependencies, views, and delegation of a given transaction are defined through statements which are attached to the corresponding method. Such statements can be set to coincide with any of the significant events of a transaction, i.e. begin, commit and abort. Therefore, the main body of KALA declarations for a given method contains `begin`, `commit`, and `abort` statements, and each of these statements contains a nested block of dependency, view and delegation statements.

A significant event is defined by the `Event` linglet.

```
Linglet Event {
   syntax {  name  "{" body "}" }
}
```

The three significant events that KALA currently supports are defined in the language specification by reusing the same `Event` linglet and binding its name to the three linglets `"begin"`, `"commit"` and `"abort"`.

```
name KALA
base KALA

Begin=Event
         name: "begin"
```

```
         body: ...
Abort=Event
         name: "abort"
         body: ...
Commit=Event
         name: "commit"
         body: ...
```

The behavior of multiple, possibly concurrent, transactions is governed by constraints between the significant events of these transactions. To facilitate reasoning about these constraints, a first abstraction was created, called dependencies. Views relax the conservative visibility between transactions, and delegation relaxes the conservative ownership of the data accessed and modified by a transaction.

C.1.5 Dependencies

A dependency is added by a triplet of the source, the type of dependency, and the target of the dependency. Source and target are transaction identifiers. A dependency statement consists of the keyword `dep`, followed by such a triplet, enclosed between parenthesis.

```
Linglet Dependency {
   syntax { "dep" "(" source operator target ")"  }
}
```

KALA supports the dependencies of the underlying transaction monitor. The transaction monitor ATPMos [Fab05] supports five dependencies: commit dependency, weak-abort dependency, begin-on-commit dependency, begin-on-abort dependency and compensation dependency. These are respectively modeled by the `CD`, `WD`, `BCD`, `BAD` and `CMD` operator. A detailed discussion about the semantics falls outside the scope of this dissertation. We refer the reader to [Fab05].

C.1.6 View

In a number of ATMS, isolation between different transactions is relaxed and it is possible for one running transaction T_i to see the results of another transaction T_j while T_j is still executing. Isolation is relaxed by the `view` statement.

The statement `view`, defined below, consists of two parts, respectively declaring the source and destination. Views are removed by prepending the destination identifier with the minus sign -.

```
Linglet View {
   syntax { "view" "(" !("-" [ ast minus:true ]) source target ")"  }
}
```

C.1.7 Delegation

The delegate statement delegates the responsibility for the accessed and change data of a transaction to another transaction.

Delegation is specified with the `del` keyword, and takes a tuple of source and target transaction identifiers, as in the dependency specification.

```
Linglet Delegation {
   syntax { "del" "(" source target ")"  }
}
```

C.1.8 Termination

Transactions are not automatically removed from the constraint model when it ends. This is because other, running, transactions may need to place dependencies on this transaction that need to remain in place even after the transaction has ended. Hence, the programmer must therefore manually declare that these references are no longer needed when this is the case. For this, KALA offers the `terminate` and `groupTerminate` statements which can be used in begin, commit or abort significant events.

```
Linglet Terminate {
   syntax { "terminate" "(" name ")"  }
}
Linglet GroupTerminate {
   syntax { "groupTerminate" "(" name ")"  }
}
```

C.1.9 Autostart

Separately from the main control flow of the application, some ATMS require to execute secondary transactions. The execution of these transactions are controlled via dependencies.

Secondary transactions are spawned by the `autostart` statement. The `arguments` are the values which are used to execute the associated method (`name`). The `variables` are the primitive values which are called by reference, such that the secondary and the enclosing transaction share their values. Finally, a list of KALA declarations can be attached to this spawned transaction.

```
Linglet AutoStart {
syntax { "autostart" "("
                        name
                        "<" arguments ("," arguments)* ">"
                        !( "(" vars ("," arguments)*)  ")" )
                        !( "{" body  "}" )
                     ")"  }
}
```

C.2 KALA Language Specification

The language specification on the next page describes the KALA language. We nested the composition of the linglets to reflect the hierarchical layout of the AST representing KALA programs.

```
name KALA
base KALA

TxDeclaration
       signature: Signature name: ID parameter: ID..
       expression: ID.
       body: Name       name: ID.  key: Expression..
             GroupAdd   name: ID.  key: Expression..
             Alias      name: ID.  key: Expression..
             GroupAlias name: ID.  key: Expression..
             Begin=Event
                name: "begin"
                body:
                   Dependency     left: ID. dependency: ID. right: ID..
                   View           source: ID.  target: ID..
                   Delegation     source: ID   target: ID..
                   Terminate      name: ID..
                   GroupTerminate name: ID..
                   Name.
                   GroupAdd..
             Commit=Event
                name: "commit"
                body: Dependency.View.Delegation.
                      Terminate.GroupTerminate.
                      Name.GroupAdd..
             Abort=Event
                name: "abort"
                body: Dependency. View. Delegation.
                      Terminate.GroupTerminate.
                      Name.GroupAdd..
             Autostart
                argument: Expression.
                name: Signature.
                body: Dependency. View. Delegation.
                      Terminate.GroupTerminate.
                      Name.GroupAdd...
```

C.3 ATMS KALA Specifications

C.3.1 Saga KALA Specification

The KALA specification to declare a Saga is given by means of the following example banking application which is discussed in Section 7.5.2.

```
class Cashier {
  public void moneyTransfer (Account from, Account to, int amount){
    this.transfer(from, to, amount);
    this.printReceipt(from, to, amount);
    this.logTransfer(from, to, amount);
  }
  private void transfer (Account from, Account to, int amount)
  { ... }
  private void printReceipt(Account from, Account to, int amount)
  { ...}
  private void logTransfer(Account from, Account to, int amount)
  { ... }
}
```

The KALA specification, shown below, declares for the above example a Saga transaction for the `moneyTransfer(...)` method comprising of three steps corresponding to the methods `transfer(...)`, `printReceipt(...)` and `logTransfer(...)`. All, except for the last step, are compensated by the secondary transaction `transfer(...)` and `printReceipt(...)` respectively, which invokes a compensating action in case the step must be roll backed.

```
Cashier.moneyTransfer(BankAccount, BankAccount, int) {
    name(self <Thread.currentThread()>)
    commit {
       groupTerminate(<""+self+"Comp">)
       groupTerminate(<""+self+"Step">)
       terminate(self)
    }
    abort {
       groupTerminate(<""+self+"Comp">)
       groupTerminate(<""+self+"Step">)
       terminate(self)
    }
}

Cashier.transfer(BankAccount, BankAccount, int) {
    alias (Saga <Thread.currentThread()>)
    groupAdd (self <""+Saga+"Step">)
    autostart (transfer(BankAccount, BankAccount, int)
```

```
       <dest, source, amount> {
          name(self <""+Saga+"Comp">)
          groupAdd(self <""+Saga+"Comp">)
       });
    begin {
       alias (Comp <""+Saga+"Comp">)
       dep(Saga ad self, self wd Saga, Comp bcd self)
    }
    commit {
       alias (Comp <""+Saga+"Comp">)
       dep(Comp cmd Saga, Comp bad Saga)
    }
}

Cashier.printReceipt(Account, Account, int) {
    alias (Saga <Thread.currentThread()>)
    alias (CompPrev <""+Saga+"Comp">)
    groupAdd (self <""+Saga+"Step">)
    autostart (Cashier.printTransferCancel(Account,Account,int,int)
       <source, dest, amount, num_receipt> (num_receipt) {
          name(self <""+Saga+"Comp">)
          groupAdd(self <""+Saga+"Comp">)
       })
    begin {
       alias (comp <""+Saga+"Comp">)
       dep(saga ad self, self wd saga, comp bcd self)
    }
    commit {
       alias (comp <""+Saga+"Comp">)
       dep(CompPrev wcd comp, comp cmd saga, comp bad saga)
    }
}

Cashier.logTransfer(BankAccount, BankAccount, int){
    alias (Saga <Thread.currentThread()>)
    groupAdd(self <""+Saga+"Step">)
    begin {
       dep(Saga ad self, self wd Saga, Saga scd self)
    }
}
```

Bibliography

[ADK+98] William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in Intentional Programming. In ICSR '98: Proceedings of the 5th International Conference on Software Reuse, page 114, Washington, DC, USA, 1998. IEEE Computer Society. 87

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). Oxford University Press, New York, August 1977. 325

[Amb02] Scott W. Ambler. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. John Wiley & Sons, Inc., New York, NY, USA, 2002. 4, 327

[ANV05] Scott W. Ambler, John Nalbone, and Michael J. Vizdos. The Enterprise Unified Process: Extending the Rational Unified Process. Prentice Hall PTR, 2005. 327

[App98] Andrew W. Appel. Modern Compiler Implementation in C. Cambridge University Press, 1998. 9, 122, 163, 247

[Ayc03] John Aycock. A Brief History of Just-in-time. ACM Comput. Surv., 35(2):97–113, 2003. 34

[BB02] Anders Berglund and Scott Boag. XML Path Language (XPath) 2.0 W3C Working Draft, November 2002. 232

[BBG+60] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alain J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph H. Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the Algorithmic Language ALGOL 60. Commun. ACM, 3(5):299–314, 1960. 11, 31, 101, 151

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, Proceedings of the Conference on Object-Oriented

Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming, pages 303–311, Ottawa, Canada, 1990. ACM Press. 223

[BCD+00] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN: User Manual. Loria, Nancy, France, v3.4 edition, January 2000. 55

[BD] Jan Bosch and Yvonne Dittrich. Domain-Specific Languages for a Changing World. 6, 85, 87

[BDKT03] Kim B. Bruce, Robert L. Scot Drysdale, Charles Kelemen, and Allen Tucker. Why math? Commun. ACM, 46(9):40–44, 2003. 2

[BFG95] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report Technical Report No. 95-373, Queen's University, Department of Computing and Information Science, January 1995. 57

[BGW93] Daniel G Bobrow, Richard G. Gabriel, and Jon L. White. CLOS in Context - The Shape of the Design Space. In Object-Oriented Programming: The Clos Perspective. MIT Press, 1993. 214

[BHW97] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In Modern Software Tools for Scientific Computing, pages 353–372. Birkhauser Boston Inc., Cambridge, MA, USA, 1997. 55

[Bie04] Celeste Biever. Language may Shape Human Thought. Science Express. Pages 1-10., August 2004. 1

[BL02] Noury Bouraqadi and Thomas Ledoux. Aspect-Oriented Programming Using Reflection. Technical Report 2002-10-3, Ecole des Mines de Douai, 2002. 328

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for Implementing Domain-specific Languages. In Proceedings Fifth International Conference on Software Reuse, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE. 27, 33, 88

[Boi06] Paul Du Bois. Game Developers Conference. Lua in Games Roundtable. 2006 https://www.cmpevents.com/GD06/a.asp?option=G&V=3&id=384048, 2006. 4

[Bos97] Jan Bosch. Delegating Compiler Objects: Modularity and Reusability in Language Engineering. Nordic Journal of Computing, 4(1):66–92, Spring 1997. 6, 7, 85, 101

[Bri05] Johan Brichau. Integrative Composition of Program Generators. PhD thesis, Vrije Universiteit Brussel, 2005. 13, 33, 67, 71, 80, 81, 124, 323, 346

[BST+94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. IEEE Softw., 11(5):89–94, 1994. 33, 80

[Bud86] Timothy Budd. A little Smalltalk. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 166

[CAE+76] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, 20(6):560–575, 1976. 118, 150

[Cam97] Grady H. Jr. Campbell. Domain-specific Engineering. In Proceedings of the Embedded Systems Conference, 1997. 4

[CB05] Thomas Cleenewerck and Johan Brichau. An Invasive Composition System for Local-to-Global Transformations. In Proceedings of LDTA 2005 (Fifth Workshop on Language Descriptions, Tools, and Applications) To appear in Electronic Notes of Theoretical Computer Science, Elsevier, pages 44–63, 2005. 51

[CDE+05] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí Oliet, José Meseguer, and Carolyn Talcott. Maude Manual (Version 2.1.1), April 2005. 24

[CDMS01] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software Engineering by Source Transformation - Experience with TXL. SCAM'01 - Int. Workshop on Source Code Analysis and Manipulation, pages 168–178, November 2001. 47

[CDMS02] James Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Grammar programming in TXL. SCAM'02 - IEEE 2nd International Workshop on Source Code Analysis and Manipulation, Montreal, October 2002. 51

[CF04] Ryan Culpepper and Matthias Felleisen. Taming macros. In Gabor Karsai and Eelco Visser, editors, GPCE, volume 3286 of Lecture Notes in Computer Science, pages 225–243. Springer, 2004. 6

[CFW85] William D. Clinger, Daniel P. Friedman, and Mitchell Wand. A Scheme for a Higher-Level Semantic Algebra, pages 237–250. Cambridge University Press, 1985. 110

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architectures, October 2003. 36

[CH05] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium, pages 1–10, New York, NY, USA, 2005. ACM Press. 322

[Cho56] Noam Chomsky. Three Models for the Description of Language. IRE Transactions on Information Theory, IT-2 no. 3:113–124, 1956. 99

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-Based Definition of Metaprogramming Systems. ACM Trans. Program. Lang. Syst., 6(1):20–54, 1984. 28

[CK07] Thomas Cleenewerck and Ivan Kurtev. Separation of concerns in translational semantics for dsls in model engineering. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, SAC, pages 985–992. ACM, 2007. 209, 324, 327

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November, 1999. 66, 67

[Cle03] Thomas Cleenewerck. Component-based DSL Development. In Proceedings of GPCE'03 Conference, Lecture Notes in Computer Science 2830, pages 245–264. Springer-Verlag, 2003. 50

[Cle05] Thomas Cleenewerck. Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System. In Proceedings of the Symposium on Applied Computing Conference, 2005. 51, 336, 343, 345

[Cod72] Edgar F. Codd. Relational Completeness of Data Base Sublanguages. In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972. 150

[CX03] Chiyan Chen and Hongwei Xi. Implementing Typeful Program Transformations. In PEPM '03: Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, pages 20–28, New York, NY, USA, 2003. ACM Press. 323

[Cza98] Krzysztof Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technical University of Ilmenau, 1998. 80

[Dea97] Terrence W. Deacon. The Symbolic Species: The Co-evolution of Language and the Brain. W.W. Norton, 1997. 1

[Dev98] Kris Devolder. Type Oriented Logic Meta Programming. PhD thesis, Vrije Universiteit Brussel, 1998. 66

[DGL+03] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, and Jim Steel. Model transformation: A declarative, reusable patterns approach. In EDOC, pages 174–185. IEEE Computer Society, 2003. 60

[Dij76] Edsger W. Dijkstra. Executional abstraction. Prentice-Hall, 1976. 5, 98

[Dij99] Edsger W. Dijkstra. Computing science: Achievements and challenges. SIGAPP Appl. Comput. Rev., 7(2):2–9, 1999. 2

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In ECOOP, pages 144–169, 2004. 159, 328

[EJ01] Rober Esser and Joern Janneck. A Framework for Defining Domain-specific Visual Languages. In Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2001. 244

[Ekm04] Torbjörn Ekman. Rewritable Reference Attribute Grammars: design, implementation and applications. Master's thesis, Lund University, 2004. 329

[Ekm06] Torbjörn Ekman. Extensible Compiler Construction. PhD thesis, Lund University, 2006. 324, 329

[EN94] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc., 1994. 150

[Ern03] Erik Ernst. Separation of Concerns. In Proceedings of Software Engineering Properties of Languages for Aspect Technologies, SPLAT 2003, in assoc. with AOSD 2003, page 6 pages, 2003. 132

[ERT99] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: language and environment. Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools, pages 551–603, 1999. 57

[Fab05] Johan Fabry. Modularizing Advanced Transaction Management. PhD thesis, Vrije Universiteit Brussel, 2005. 265, 266, 267, 269, 278, 279, 306, 351, 354

[Fai98] Rickard E. Faith. Debugging Programs after Structure-Changing Transformation. PhD thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, 1998. 327

[FC05] Johan Fabry and Thomas Cleenewerck. Aspect-Oriented Domain Specific Languages for Advanced Transaction Management. In Proceedings of the International Conference on Enterprise Information Systems, pages 428–432, 2005. 270

[Fel91] Matthias Felleisen. On the Expressive Power of Programming Languages. Science of Computer Programming, 17(1–3):35–75, December 1991. 3, 4, 24, 26, 101, 106, 207

[FK92] Robert G. Fichman and Chris F. Kemerer. Object-Oriented and Conventional Analysis and Design Methodologies. Computer, 25(10):22–39, 1992. 30

[FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new Graph Transformation Language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, Proc. Theory and Application to Graph Transformations (TAGT'98), Paderborn, volume 1764 of LNCS. Springer, November 1998. 57

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, June 2005. 29, 31

[FP88] Alan Ford and F. David Peat. The Role of Language in Science. Foundations of Physics, 18(1233), 1988. 1

[FP06] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In Peri L. Tarr and William R. Cook, editors, OOPSLA Companion, pages 855–865. ACM, 2006. 5

[Fra02] David Frankel. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons, Inc., New York, NY, USA, 2002. 326

[Gab91] Richard P. Gabriel. LISP: Good news, bad news, how to win big. AI Expert, 6(6):30–39, June 1991. 24

[GAS98] Stefanos Gritzalis, George Aggelis, and Diomidis Spinellis. Programming Languages for Mobile Code: A Problems Viewpoint. In Proceedings of the First International Network Conference INC '98, pages 210–217. IEE, Internet Research, July 1998. 322

[GdM03] Jeremy Gibbons and Oege de Moor. The Fun of Programming. Palgrave Macmillan, March 2003. 91

[Ghu06] Abdulaziz Ghuloum. An Incremental Approach to Compiler Construction. Scheme and Functional Programming Workshop 2006 - Report, 2006. 98

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pages 249–259, New York, NY, USA, 1987. ACM Press. 267

[Gor04] Peter Gordon. Numerical Cognition Without Words: Evidence from Amazonia. Science Express, October 2004. 1

[Gra94] Paul Graham. On Lisp: advanced techniques for Common Lisp. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994. 5, 169, 171

[Gro92] Josef Grosch. Multiple Inheritance in Object-Oriented Attribute Grammars. Technical Report Compiler Generation Report 28, GMD Karlsruhe, Februari 1992. 74, 76, 155

[Gro02a] Object Management Group. Meta Object Facility (MOF) Specification 1.4., 2002. 326

[Gro02b] Object Management Group. Object Management Group. MOF 2.0 Query/Views/Transformations RFP, 2002. 324

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Vincenzo Ambriola and Genoveffa Tortora, editors, Advances in Software Engineering and Knowledge Engineering, pages 1–39. World Scientific Publishing Company, Singapore, 1993. 30, 153

[GST01] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In International Conference on Functional Programming, pages 74–85, 2001. 5

[Ham] Graham Hamilton. Evolving the Java Platform. Sunday Times. May 15, 2005. 4

[HE92] H. Kreowski H. Ehrig, A. Habel. Introduction to Graph Grammars with Applications to Semantic Networks. International Journal of Computers and Mathematical Applications, 23(6-9):557–572, 1992. 61

[Hed89] Görel Hedin. An Object-Oriented Notation for Attribute Grammars. In ECOOP, pages 329–345, 1989. 74

[Hed92] Görel Hedin. Incremental Semantic Analysis. PhD thesis, Lund University, Lund, Sweden, 1992. 74, 77, 155

[Hed99] Görel Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, Second Workshop on Attribute Grammars and their Applications, WAGA'99, pages 153–172, Amsterdam, The Netherlands, 1999. INRIA Rocquencourt. 73, 78, 155

[HHKR89] Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF - Reference Manual. SIGPLAN Notices, 24(11):43–75, 1989. 101

[HM03] Görel Hedin and Eva Magnusson. JastAdd: An Aspect-oriented Compiler Construction System. Sci. Comput. Program., 47(1):37–58, 2003. 6, 74, 76, 77, 155

[HO93] William Harrison and Harold Ossher. Subject-oriented Programming: A Critique of Pure Objects. SIGPLAN Not., 28(10):411–428, 1993. 80

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, pages 2–1–2–26, New York, NY, USA, 2007. ACM Press. 4

[Jan96] Theo M. V. Janssen. Compositionality. In Johan van Benthem and Alice ter Meulen, editors, Handbook of Logic and Language, pages 417–473. Elsevier, Amsterdam, 1996. 99, 101, 103

[JD93] Mark P. Jones and Luck Duponcheel. Composing monads. Technical report, 1993. 26

[Jef] Ron Jeffries. Thoughts and Actions: Beyond Haskell. http://www.xprogramming.com/xpmag/dbcBeyondHaskell.htm, July, 2006. 2

[Jet] JetBrains. Meta-Programming System. 2005. 31

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Proceedings of Model Transformations in Practice Workshop, part of the MoDELS 2005 Conference, October 2005. 324

[JKN95] Esa Jarnvall, Kai Koskimies, and Maarit Niittymaki. Object-oriented Language Engineering with TaLE. Object Oriented Systems, pages 77–98, 1995. 86, 98

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In UNIX Programmer's Manual, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. 8, 28

[JY01] Gilt Joseph and Tsoglin Yuri. JAMOOS A Domain-Specific Language for Language Processing. Journal of Computing and Information Technology, pages 305–321, september 2001. 86, 98, 101, 332

[KC86] Setrag N. Khoshafian and George P. Copeland. Object Identity. In Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications, 1986. 175

[Kel07] Andy Kellens. Co-design and co-evolution of source code and its structural regularities using Intensional Views. PhD thesis, Vrije Universiteit Brussel, 2007. 77

[Kic01] Gregor Kiczales. The Future of Reflection. Invited talk at the Third Interna- tional Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflec- tion 2001), September 2001., 2001. 328

[KK83] Paul Kay and Willett Kempton. What is the Sapir-Whorf hypothesis? Technical Report COGSCI-83-08, 1983. 1

[KLG91] Simon M. Kaplan, Joseph P. Loyall, and Steven K. Goering. Specifying Concurrent Languages and Systems with D-grammars. In Fourth International Workshop on Graph Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 475–489. Springer Verlag, 1991. 61

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. 78, 322, 328

[Klo92] Jan W. Klop. Term Rewriting Systems. In Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), volume 2. Clarendon, 1992. 8, 47

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. Mathematical Systems Theory, 2(2):127–145, 1968. 8, 72, 325

[KRB91] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, 1991. 214

[Kru00] Philippe Kruchten. The Rational Unified Process: An Introduction, Second Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 327

[KV01] Tobias Kuipers and Joost Visser. Object-oriented Tree Traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, Electronic Notes in Theoretical Computer Science, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA). 33

[KW94] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. Acta Informatica, 31(7):601–627, 1994. 99, 328

[Lai] Cameron Laird. XSLT Powers a New Wave of Web Applications. http://www.linuxjournal.com/article/5622, 2002. 226

[Lan66] Peter J. Landin. The next 700 programming languages. Commun. ACM, 9(3):157–166, 1966. 3, 24

[Leh96] Manny M. Lehman. Laws of Software Evolution Revisited. In EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology, pages 108–124, London, UK, 1996. Springer-Verlag. 4, 324, 327

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. IEEE Softw., 6(5):38–48, 1989. 70

[Lie96] Karl J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, 1996. 70

[LV03] Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In Proc. of PADL'03, January 2003. 91

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, pages 147–155, New York, NY, USA, 1987. ACM Press. 214

[MB97] Kai-Uwe Mätzel and Walter R. Bischofberger. Designing Object Systems for Evolution. Theor. Pract. Object Syst., 3(4):265–283, 1997. 143, 175

[Mic] Sun Microsystems. New features and enhancements j2se 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html. 5

[MM] Jishnu Mukerji and Joaquin Miller. Model Driven Architecture, http://www.omg.org/cgibin/doc?ormsc/2001-07-01, July 2001. 326

[Mog97] Eugenio Moggi. Metalanguages and applications. In Andrew M. Pitts and Peter Dybjer, editors, Semantics and Logics of Computation, volume 14, pages 185–239. Cambridge University Press, Cambridge, 1997. 26

[Mon74] Richard Montague. Universal grammar. In Richmond Thomason, editor, Formal Philosophy: Selected Papers of Richard Montague, pages 222–246. Yale University Press, New Haven, CN, 1974. 101, 103

[Mos81] J. Eliot B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, Cambridge, MA, USA, 1981. 267

[Mos04] Peter Mosses. Modular Language Descriptions, volume 3286/2004, pages 489–490. Springer Berlin / Heidelberg, 2004. 45

[MuLA99] Marjan Mernik, Viljem Žumer, Mitja Lenič, and Enis Avdičaušević. Implementation of Multiple Attribute Grammar Inheritance in the Tool LISA. SIGPLAN Not., 34(6):68–75, 1999. 74, 78, 155, 328, 329

[New] Jeff Newbern. All about Monads. http://www.nomaware.com/monads/html/. 91, 122, 338

[Nie01] Lasse R. Nielsen. A Selective CPS Transformation. Electr. Notes Theor. Comput. Sci., 45, 2001. 110

[OdMS00] Kevin Backhouse Oege de Moor and S. Doaitse Swierstra. First Class Attribute Grammars. Informatica: An International Journal of Computing and Informatics, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications. 6, 9, 74, 77, 329

[OKK+96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying Subject-oriented Composition. Theor. Pract. Object Syst., 2(3):179–202, 1996. 33, 80, 238, 244

[OV05] Karina Olmos and Eelco Visser. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules, volume 3443/2005 of Lecture Notes in Computer Science, pages 204–220. Springer Berlin / Heidelberg, 2005. 329

[Paa95] Jukka Paakki. Attribute Grammar Paradigms. A High-level Methodology in Language Implementation. ACM Comput. Surv., 27(2):196–255, 1995. 6, 7, 8, 28, 72, 76, 138, 155

[Pae93] Andreas Paepcke. User-Level Language Crafting. In Object-Oriented Programming : the CLOS perspective, pages 66–99. MIT Press, 1993. 219

[Par72] David L. Parnas. A Technique for Software Module Specification with Examples. Communications of the ACM, 15(5):330–336, May 1972. 45

[PR04] Magnus Petersson and Thomas Raneland. Java 1.5 as modular language extensions, 2004. 324

[PtMW90] Barbara H. Partee, Alice ter Meulen, and Robert E. Wall. Mathematical Methods in Linguistics. Kluwer, Dordrecht, 1990. 104

[Rey93] John C. Reynolds. The Discoveries of Continuations. LISP and Symbolic Computation, 6(3–4):233–247, 1993. 110

[SAA99] S. Doaitse Swierstra and Pablo R. Azero Alcocer. Fast, Error Correcting Parser Combinators: A Short Tutorial. In Jan Pavelka, Gerard Tel, and Miroslav Bartosek, editors, SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, volume 1725 of LNCS, pages 111–129, November 1999. 217, 247

[SB98] Yannis Smaragdakis and Don S. Batory. Implementing Layered Designs with Mixin Layers. In ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, pages 550–570, London, UK, 1998. Springer-Verlag. 80

[Sch91] Andy Schürr. PROGRESS: A VHL-Language Based on Graph Grammars. In Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science, pages 641–659, London, UK, 1991. Springer-Verlag. 57

[SD02] Vytautas Stuikys and Robertas Damasevicius. Taxonomy of the Program Transformation Processes. Information technology and control, Kaunas, Technologija, 22(1):39 – 52, 2002. 36

[Ser99] Manuel Serrano. Wide Classes. In European Conference on Object-Oriented Programming, ECOOP'99, June 1999. 159

[Sim95a] Charles Simonyi, 1995. 6

[Sim95b] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In The Future of Software, Proceedings of the Joint International Computers Limited/University of Newcastle Seminar, University of Newcastle, 1995. 31, 87

[Sim96] Charles Simonyi. Intentional Programming - Innovation in the Legacy Age, IFIP WG 2.1 meeting, Microsoft Research, 1996. 6, 31, 87

[SLB+99] Marcelo Sant'Anna, Julio Leite, Ira Baxter, Dave Wile, Ted Biggerstaff, Don Batory, Prem Devanbu, and Liz Burd. International workshop on software transformation systems (sts '99). In ICSE '99: Proceedings of the 21st international conference on Software engineering, pages 701–702, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. 28

[Smi84] Brian Cantwell Smith. Reflection and Semantics in LISP. In Principles of programming languages (POPL84), January 1984. 214

[Spe04] Elizabeth Spelke. Which Comes First, Language or Thought? Harvard University Gazette, 2004. 1

[Ste94] Patrick Steyaert. Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks. PhD thesis, Vrije Universiteit Brussel, 1994. 214, 222

[Ste99] Guy L. Jr. Steele. Growing a language. Higher-Order and Symbolic Computation, 12(3):221–236, 1999. 4

[Sul01a] Gregory T. Sullivan. Aspect-oriented Programming Using Reflection and Metaobject Protocols. Commun. ACM, 44(10):95–97, 2001. 328

[Sul01b] Gregory T. Sullivan. Aspect-Oriented Programming using Reflection. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Oct. 14-18, 2001, Tampa Bay, Florida, USA, 2001. 328

[Swi] S. Doaitse Swierstra. Attribute grammars: A Short Tutorial. Dept of Computer Science Utrecht University, 2003. xvii, 6, 74

[SWW+88] V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic Analysis in a Concurrent Compiler. In PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, pages 233–240, New York, NY, USA, 1988. ACM Press. 8, 30

[SZJ02] Myat Swe Soe, Hongyu Zhang, and Stan Jarzabek. XVCL: A Tutorial. In Proc. 14 th Int. Conf. on Software Engineering and Knowledge Engineering SEKE02, pages 341–349. ACM Press, 2002. 66

[TB03] Dave Thomas and Brian M. Barry. Model Driven Development: The Case for Domain Oriented Programming. In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 2–7, New York, NY, USA, 2003. ACM Press. 4, 326

[Ten76] R. D. Tennent. The denotational semantics of programming languages. Commun. ACM, 19(8):437–453, 1976. 108

[Ten91] R.D. Tennent. Semantics of Programming Languages. Prentice-Hall, 1991. TEN r 91:1 1.Ex. 102

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In International Conference on Software Engineering, pages 107–119, 1999. 45

[vdBK02] Mark G.J. van den Brand and Paul Klint. ASF+SDF Meta-Environment User Manual. Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, July 2002. 50, 169, 171, 322

[vDHK96] Arie van Deursen, Jan Heering, and Paul Klint, editors. Language Prototyping: An Algebraic Specification Approach, volume 5 of AMAST Series in Computing. World Scientific Publishing Co., 1996. 47

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. SIGPLAN Notices, 35(6):26–36, 2000. 4, 27, 28, 33, 184, 244

[Vel03] Velocity. Velocity 1.3.1, The Apache Jakarta Project, March 2003, http://jakarta.apache.org/velocity/, 2003. 66, 67, 169, 171

[Vis01a] Eelco Visser. Scoped Dynamic Rewrite Rules. Electronic Notes in Theoretical Computer Science, 59(4), 2001. 54, 324, 329

[Vis01b] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. Lecture Notes in Computer Science, 2051:357–361, 2001. 47, 55, 329

[VKV03] Mark G. J. Van Den Brand, Paul Klint, and Jurgen J. Vinju. Term Rewriting with Traversal Functions. ACM Trans. Softw. Eng. Methodol., 12(2):152–190, 2003. 9, 52

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pages 131–145. ACM Press, 1989. 73

[vW03] Jonne van Wijngaarden. Code Generation from a Domain Specific Language Designing and Implementing Complex Program Transformations. Master's thesis, Universiteit Utrecht, July 2003. 138, 329

[VWKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In European Conference on Object Oriented Programming (ECOOP), Lecture Notes in Computer Science. Springer Verlag, July 2007. To Appear. 324

[vWV03] Jonne van Wijngaarden and Eelco Visser. Program Transformation Mechanics. Technical Report UU-CS-2003-048, Universiteit Utrecht, 2003. 36, 39, 43, 67, 69, 101

[Wad92] Philip Wadler. Comprehending Monads. Mathematical Structures in Computer Science, 2(4), 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.). 91

[Wan84] Mitchell Wand. A Semantic Prototyping System. In SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction, pages 213–221, New York, NY, USA, 1984. ACM Press. 28

[WdMBK02] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In Proc. 11th International Conf. on Compiler Construction, volume 2304 of Lecture Notes in Computer Science. Springer-Verlag, 2002. 9, 76, 78, 325

[WdMS+01] Erik Van Wyk, Oege de Moor, Ganesh Sittampalam, Ivan Sanabria-Piretti, Kevin Backhouse, and Paul Kwiatkowski. Intentional Programming: A Host of Language Features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001. 87, 200

[WGRM05] Hui Wu, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Weaving a Debugging Aspect into Domain-Specific Language Grammars. In ACM Symposium for Applied Computing (SAC) - Programming for Separation of Concerns Track, pages 1370–1374, 2005. 327

[Whi02] Ray Whitmer. Document Object Model (DOM) Level 3 XPath Specification W3C, March 2002. 9

[Who56] Benjamin Whorf. Language, Thought, and Reality: Selected Writings. MIT Press, Boston, MA, 1956. 1