Reusability for Mechanized Meta-Theory

Steven Keuchel

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Supervisor:
prof. dr. ir. Tom Schrijvers

Department of Applied Mathematics, Computer Science and Statistics
Faculty of Sciences, Ghent University

Acknowledgments

Performing my research and writing this thesis has been a long and enriching journey, and a countless number of people have in many different ways had an impact on my work and on me during that time. I want to thank each one of you. Unfortunately I cannot possibly remember and list everybody here, but I would like nevertheless to thank several people explicitly.

I first wish to acknowledge my advisor, Tom Schrijvers. I have been very fortunate to have your help in preparing my publications and presentations, and your unyielding support and patience. I can safely say, that without you I would not have finished my Ph.D. and this thesis and I will be forever grateful that you got me here.

After finishing a provisional manuscript of this thesis, it was sent to an "examination board" to evaluate the quality of my research and my contributions. I am thankful to Christophe Scholliers, Eric Laermans, François Pottier, Peter Dawyndt and Stephanie Weirich for accepting this task, and for the valuable feedback and comments provided. I would likewise thank Georgios Karachalias, Klara Marntirosian and Marie-Anne Haure for proofreading parts of earlier drafts.

My interest in the subject of my dissertation was formed during my time in Utrecht, long before I started pursuing my Ph.D. in Ghent. I thank José Pedro Magalhães, Johan Jeuring and Andres Löh for introducing and teaching me about datatype-generic programming, dependently typed programming, proof assistants and programming language meta-theory. I would especially like to thank and curse Andres for getting me hooked on the problem of variable binding. It is a fascinating topic and I still take pleasure working on it, but your warnings were clearly justified.

Over the years, I have enjoyed collaborating and exchanging ideas with several excellent researchers which were a great inspiration for this thesis. During my research visit at Penn, Stephanie Weirich was bombarding me with binding related ideas, requirements and special cases. In our discussions you were always energetic and full of excitement about this topic, and this was one of the best work experiences that I had during my Ph.D. for which I thank you. I think by now I know the answer to a lot of open questions that we had at the time, but I seem to always find new questions that I would love to talk to you about. I would likewise thank Arthur Azevedo de Amorin, Dominique Devriese, François Pottier and Randy Pollack for our interesting discussions about variable binding at conferences or elsewhere and for their interest in my work.

Even though we lost contact over the years I would like to thank Christopher Schwaab for introducing me to effect handlers early in my Ph.D. and his input on effect modularization. This is exactly the topic that I worked on with Benjamin Delaware that jump started my endeavor to mechanize meta-theory. This was the first time that I used Coq and I learned a lot from you through our joint project for which I thank you.

Of course also thanks to my past and present colleagues from my research group: Alexander, Amr, Benoit, George, Gert-Jan, Klara, Maciej, Paolo, and Ruben. We certainly had a lot of fun, and some serious and not so serious discussions about life, the universe and everything.

I thank my family and friends for their incredible support and the good times out side of work; in particular, my (ex-)flatmate Pablo who introduced me to good gin and good coffee and lots of places in Ghent that I would otherwise not have visited, and George who left me and others always amazed and who was my mental safe haven. Last and perhaps most of all, thanks to my girlfriend Marie-Anne for her love, patience, and encouragement during the last years. You're the best.

Steven Keuchel
May 2018

Summary

Computer scientists develop new programming languages and improve existing ones to let us write better programs faster. One goal is to develop languages with useful meta-theoretical properties, like, for example, safety guarantees for all programs expressed in the language. These guarantees are useful when reasoning about programs: a programmer can use them when assessing correctness of her program, and a compiler can use them for optimizations.

Sadly, developing languages is difficult: different language features often display complicated interactions in edge cases, which are easily overlooked, and can thus invalidate assumptions about meta-theoretical properties with potentially disastrous implications. Therefore researchers began to formally specify programming languages and verify their meta-theory in proof assistants, a process which is also known as mechanization. Unfortunately, mechanization of meta-theory is not common practice because of its steep learning curve and large development effort. As a result, if done at all, mechanizations often only cover a manageable subset of the language with the downside that results do not always carry over to the full language.

To increase the adoption of mechanization and to further scale it to realistic programming languages, it is imperative to reduce the costs. This thesis investigates code reuse as a means to achieve this, specifically principled reuse via modularity and genericity which we discuss in turn.

Modularity Different programming languages often have features in common, for instance boolean or exception handling. The first part of this thesis deals with reuse by modularly sharing specification, implementation and meta-theoretic proofs of features by multiple languages.

A stumbling block is that inductive definitions and proofs are closed to extension. This is solved by using datatype-generic programming techniques to modularize datatypes, semantics functions and inductive proofs. A case study shows the advantages of our approach over an existing solution.

Modularizing proofs about languages with side effects is exceptionally challenging, since the theorem usually depend on all the effects the language uses and side effectful features display a lot of interaction. We improve this situation by developing a new denotational semantics based on monadic interpreters that allows us to factor the type safety theorem into seperate parts: a feature theorem that captures the well-typing of monadic denotations of an individual feature, and an effect theorem that adapts the well-typing of denotations to a fixed superset of effects. The type safety proof for a particular language combines the theorems for all its features with the theorem of all its effects. Our case study shows the effictiveness of our approach by modularizing five language features, including three with effects.

While our techniques achieve the intermediate goals of modularization and reuse, the complexity and bookkeeping involved inflate the overall development effort. Further research and direct integration into proof assistants is needed to make the techniques practical.

Genericity Nearly every high-level programming language uses variable binding in its syntax. The operational semantics of such languages often implements reduction of constructs, that involve binding, by means of variable substitution. Meta-theoretic proofs need to deal with properties of this substitution. The substitution function and the proofs of its properties can be considered boilerplate, since they follow a pattern that only depends on the syntax and the scoping rules of the language. This boilerplate can represent a large part of the whole mechanization and should therefore best be taken care of automatically. The second part of this thesis develops a generic solution to this problem.

We develop a new declarative language Knot for the specification of abstract syntax with variable binding, and for semantic relation on top of the syntax. A type system ensures that expressions in the definition of relations are always well-scoped. We give an interpretation of Knot specifications using de Bruijn terms which we also implemented as a datatype generic library Loom in Coq. Boilerplate lemmas are implemented by generic elaboration functions into domain-specific witness languages. In particular, to the best of our knowledge, we are the first to provide elaborations for shifting and substitution lemmas of semantic relations using a first-order approach. We formally

proof the correctness of the elaborations and the soundness of the witness languages.

For practical mechanizations, we developed the Needle code generator that compiles a Knot language specification into Coq definition for that language including variable binding boilerplate. Needle's core proof elaboration functions are Haskell ports of the verified Loom functions which boosts our confidence in the correctness of Needle.

Our evaluation shows substantial savings in comparison to fully manual Coq mechanizations of type safety for various calculi. In particular, our solution to the POPLmark challenge (1a + 2a) is by far the shortest compared to other approaches.

In conclusion, this thesis extends upon existing work and provides novel insights into code reusability for mechanization of meta-theory and thereby takes another step to scaling these methods to realistic programming languages.

Samenvatting

Computerwetenschappers ontwikkelen nieuwe programmeertalen en verbeteren bestaande talen om ons sneller betere programma's te laten schrijven. Een doel is om talen te ontwikkelen met bruikbare meta-theoretische eigenschappen, zoals bijvoorbeeld veiligheidsgaranties voor alle programma's uitgedrukt in de taal. Deze garanties zijn nuttig omdat een programmeur ze bijvoorbeeld kan gebruiken om de correctheid van haar programma's te beoordelen, en omdat een compiler ze kan gebruiken voor optimalisaties.

Helaas is het ontwikkelen van talen moeilijk: taalconcepten hebben vaak ingewikkelde onderliggende interacties in randgevallen die gemakkelijk te missen zijn, en fouten kunnen aannames over meta-theoretische eigenschappen tenietdoen, met mogelijk ernstige gevolgen. Daarom zijn onderzoekers begonnen met het formeel specificeren van programmeertalen en met het verifiren van hun meta-theorie in bewijsassistenten; dit proces wordt mechanisatie genoemd.

Helaas heeft mechanisatie een steile leercurve en hoge ontwikkelingskosten en is daarom geen gangbare praktijk. Als mechanisatie wordt gebruikt, dan meestal alleen op een beperkt deel van de taal, met het nadeel dat resultaten niet altijd kunnen overgedragen worden naar de volledige taal.

Om het gebruik van mechanisaties uit te breiden en om realistische talen binnen handbereik te brengen, is het noodzakelijk om de kosten te reduceren. Dit proefschrift onderzoekt hergebruik van code als middel om dit te bereiken, en in het bijzonder het principieel hergebruik via modulariteit en genericiteit.

Modulariteit Verschillende programmeertalen hebben vaak gemeenschappelijke concepten zoals booleaanse waarden of afhandeling van excepties. Het eerste deel van dit proefschrift gaat over het modulair delen van specificatie,

implementatie en meta-theoretische bewijzen van concepten tussen meerdere talen.

Een struikelblok is dat inductieve definities en bewijzen voor extensies gesloten zijn. Dit wordt opgelost door het gebruik van datatype-generieke programmeertechnieken om datatypes, semantiek en bewijzen te modulariseren. Een case study toont de voordelen van onze aanpak ten opzichte van een bestaande oplossing.

Het modulariseren van talen met effecten is uitzonderlijk uitdagend, omdat meta-theoretische stellingen meestal afhangen van alle effecten van de taal, en concepten met effecten vaak veel interactie vertonen.

We verbeteren deze situatie door het ontwikkelen van een nieuwe denotationele semantiek gebaseerd op monadische interpreters waarmee we het typeveiligheidsbewijs kunnen opsplitsen: concept-stellingen die goed getypeerde monadische denotaties van individuele concepten bepalen, en effect-stellingen die goed getypeerde denotaties aanpassen naar een vaste verzameling van effecten. Het typeveiligheidsbewijs voor een bepaalde taal combineert de stellingen van al zijn concepten met de stelling van al zijn effecten. Onze case study toont de effectiviteit van onze aanpak door het modulariseren van vijf taalfuncties, waaronder drie met effecten.

Terwijl onze technieken de tussentijdse doelen van modularisering en hergebruik bereiken, hebben complexiteit en boekhouding de totale kosten verhoogd. Verder onderzoek en directe integratie in proefassistenten is nodig om de technieken praktisch te maken.

Genericiteit Bijna elke hoog-niveau programmeertaal maakt gebruik van variabelen in zijn syntaxis. De operationele semantiek van dergelijke talen implementeert reductie van taalconstructies met variabelen meestal door substituties van variabelen. Meta-theoretische bewijzen moeten met eigenschappen van deze substitutie redeneren. De substitutiefunctie en de bewijzen van zijn eigenschappen kunnen als boilerplate geclassificeerd worden, omdat ze een patroon volgen dat alleen afhangt van de syntaxis en de scoping regels van de taal. Deze boilerplate kan een groot deel van de hele mechanisatie uitmaken, en moet daarom het best automatisch beschikbaar zijn. Het tweede deel van dit proefschrift ontwikkelt een generieke oplossing voor dit probleem.

We ontwikkelen een nieuwe declaratieve taal Knot voor de specificatie van abstract syntaxis met variabelen, en voor semantische relaties boven op de syntaxis. Een typesysteem zorgt ervoor dat variabelen in uitdrukkingen in de definitie van relaties altijd in hun bereik worden gebruikt. We geven een interpretatie van Knot-specificaties met behulp van de Bruijn termen die we

ook gemplementeerd hebben als een datatype generieke bibliotheek Loom in Coq.

Boilerplate lemmas zijn gemplementeerd door generieke uitwerkingsfuncties naar domeinspecifieke getuige-talen. In het bijzonder, zijn wij, voor zover we weten, de eersten die uitwerkingen van verzwakking and substitutie lemmas van semantische relaties met een eerste-orde aanpak verschaffen. We bewijzen formeel de correctheid van de uitwerkingen en de deugdelijkheid van onze domeinspecifieke talen.

Voor praktische mechanisaties hebben we de Needle-codegenerator ontwikkeld die Knot-specificatie compileert naar Coq-definities inclusief variabelen boilerplate. Needle's uitwerkingsfuncties zijn Haskell-vertalingen van de geverifieerde Loom-functies; dit versterkt ons vertrouwen in de correctheid van Needle.

Onze evaluatie toont aanzienlijke besparingen in vergelijking met handmatige Coq mechanisaties van typeveiligheid voor verschillende talen. In het bijzonder is onze oplossing van de POPLmark-challenge (1a + 2a) veruit de kortste in vergelijking met andere oplossingen.

Samenvattend breidt dit proefschrift bestaand werk uit en biedt het nieuwe inzichten in het codeherbruik voor mechanisaties van meta-theorie en zet daarmee een stap naar het toepassen van deze methoden op realistische programmeertalen.

Contents

List of Publications

Keuchel, S. and Jeuring, J. T. (2012). Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming,* WGP '12, pages 57–68. ACM. Copenhagen, Denmark, September 12, 2012.

Delaware, B., Keuchel, S., Schrijvers, T., and Oliveira, B. C. d. S. (2013). Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 319-330. ACM. Boston, Massachusetts, USA, September 25–27, 2013.

Keuchel, S. and Schrijvers, T. (2013). Generic Datatypes à la Carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, WGP 13, pages 13-24. ACM. Boston, Massachusetts, USA, September 28, 2013.

Keuchel, S., Weirich, S., and Schrijvers, T. (2016). Needle & Knot: Binder Boilerplate Tied Up. In Thiemann, P., editor, *Proceedings of the 25th European Symposium on Programming,* ESOP'16, volume 9632 of *Lecture Notes in Computer Science*, pages 419–445. Springer. Eindhoven, The Netherlands, April 2–8, 2016.

Devriese, D., Patrignani, M., Piessens, F., and Keuchel, S. (2017). Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science*, Volume 13, Issue 4.

List of Figures

List of Tables

Chapter 1

Introduction

The concept of *programming* can be defined as explaining to a computer how to perform a particular task. The language of communication is known as a *programming language* and the explanation expressed in a programming language is called a *program*; collectively programs are known as *software*. Unfortunately, programming is notoriously error-prone and software often unreliable. Given the ubiquity of computers and the pervasive use of software in present-day society, this unreliability is a very costly and sometimes critical issue.

A prominent way to address this situation is to improve the programming process. In particular, it is important that the programmer can verify whether the program she has written implements the intended tasks. This is usually achieved by reasoning with the *expected behaviour* of a program, which of course requires a solid understanding of the programming language in which the program is expressed.

Programming Language Specifications In practice most programming languages do not start out as well-defined entities with an explicit meaning. Instead they are usually indirectly defined in terms of a software artifact like a compiler or interpreter, written in an existing language, that processes the programs of the new language according to some implicit intended meaning. Only after widespread use and when the implicit definition becomes untenable, a language may go through a standardization process where multiple stakeholders develop a common specification of the language.

Such a language specification has many advantages. For instance, it allows different parties to develop software tools that process programs consistently, and programmers to switch between tool vendors without a hitch. Also, it allows programmers to resolve ambiguities when reasoning about programs independent of a particular implementation.

Programming Language Theory The specification of programming languages is subject to scientific study in the field of *programming language theory*. This field deals with all aspects of language specifications: the design and implementation of their syntax and semantics, and that of auxiliary systems such as type systems. In addition the field also studies the *meta-theoretical properties* (or meta-theory for short) of languages. These meta-theoretical properties capture expectations and coherence aspects of newly developed languages or language features, such as useful safety guarantees that hold for all programs expressed in the language.

A prominent meta-theoretical property is *type safety*, i.e., the absence of dynamic type errors during execution. Because meta-theoretical properties like type safety do not automatically hold for all programming languages, we need to verify whether they actually hold for given languages. If they do not, this often points to a flaw in the language's design or a misunderstanding of the language's workings.

Mechanised Meta-Theory of Calculi Verifying meta-theoretical properties of programming languages is not a trivial activity. Due to many subtle details and edge cases, proofs can easily become unwieldy and getting the proof structure right requires a lot of effort. In order to make the process more manageable, the field of programming language theory has adopted several methodological measures.

Firstly, because full programming languages are too large to handle, meta-theoretical analysis usually restricts itself to a subset of the language, known as a *calculus*, that contains the main features of interest for the property at hand. Because calculi are much smaller so are the proofs of their meta-theory. The downside is that results for a calculus do not always carry over to the full language. Problems in the calculus often reveal problems in the full language, but the absence of problems in the calculus does not guarantee the same for the full language. For example, the Java programming language was long believed to be type-safe[1] but the addition of generics made Java's type system

[1] With dynamic type checking to ensure safety of deliberate defects like the co-variance of arrays.

unsound [Amin and Tate, 2016]. This fact was not apparent for many years, and contradicts the type safety results for several generic Java calculi [Igarashi et al., 2001; Cameron et al., 2008]. Luckily generics were never integrated into the Java Virtual Machine (JVM) which compiles checks that catch this unsoundness at runtime and throw an exception. Otherwise this unsoundness could have been used to gain direct access to the raw memory representation of objects and be used as a security exploit [Tate, 2017].

Secondly, because pen-and-paper proofs are very error-prone and human reviewers are not perfect at vetting them, meta-theoretical proofs are often written in formal languages that can be automatically checked by software tools known as *proof assistants*. This process, known as "mechanization", greatly increases the confidence in the validity of meta-theory proofs. Unfortunately, mechanization does not address the large effort of proving properties, but rather aggravates it as every little detail has to be spelled out.

Research Question Despite the mitigating efforts of the current state of the art, neither the formal specification of programming languages nor their rigorously mechanised meta-theoretical analysis are a widespread practice. The development costs are still too steep to make this possible for realistic programming languages. This leads us to the research question of this thesis:

> **How can we reduce the cost of mechanising the formal meta-theory of programming languages?**

The main approach put forward in this thesis is *reuse*. Reuse is a common approach in software engineering to reduce development cost and increase both quality and reliability. The idea is to identify functionality or patterns that are in common between different software systems, and to implement them only once in a manner that can be shared by the different software systems and reused in the development of new systems. We apply the same idea to programming language meta-theory, identifying repeated functionality and patterns, and implementing them only once in a way that can be used across proofs for different languages.

The remainder of this chapter provides a more detailed introduction to the established methodology for mechanising the meta-theory of programming languages[2] and points out opportunities for reuse.

[2] We refer the interested reader to introductory textbooks on the matter (e.g., [Pierce, 2002]) for more detail.

1.1 Programming Language Specifications

Specifications of programming languages differ in detail and precision. Industrial specifications of major programming languages are usually written in natural language and cover every aspect of the language in detail. Despite the fact that natural language leaves opportunities for ambiguity, elaborate industrial specifications provide a good reference point for language users and implementors. However, for meta-theoretical analyses more rigorous specifications are necessary. For this purpose we use *formal specifications* and a mathematical language to describe programming languages. This provides the necessary precision and avoids the ambiguities of natural languages.

This section explains necessary fundamental concepts for the formal specification of programming languages by example of a small language $\lambda_{\mathbb{B}}$: a simply-typed lambda calculus with booleans. We specify the *abstract syntax*, *static type system* and *evaluation* of $\lambda_{\mathbb{B}}$ using inductive definitions. Along the way, we define the terminology and notational conventions and make their meaning precise.

1.1.1 Syntax

The syntax of $\lambda_{\mathbb{B}}$ is given in Figure 1.1. We use a convention that is close to standard (extended Backus-Naur form) grammars. Elided in the grammar are syntactic constructs like parentheses. Yet we use parentheses freely to resolve ambiguities in terms even if the grammar does not define them. Any implementation that deals with the concrete syntax of a programming language has of course to be more rigorous.

The grammar in Figure 1.1 defines several *syntactic sorts* for $\lambda_{\mathbb{B}}$. The *meta-variable* e stands for expressions of $\lambda_{\mathbb{B}}$ of which there are 6 different kinds. An expression can either be a boolean constant **true** or **false**, a conditional form **if** e_c **then** e_t **else** e_e, an *object-language variable* (represented by the meta-variable x), the definition of a function as a λ-abstraction ($\lambda x : \tau.e$) or the application of an expression e_1 to another expression e_2. In the case of an abstraction ($\lambda x : \tau.e$) the *scope* of the variable x is the body of the abstraction e. We will also say that x is bound in e and more generally that this construct and the $\lambda_{\mathbb{B}}$ language itself use *variable binding*.

Of course, we only want to apply expressions e_1 that represent functions: either by being a λ-abstraction or evaluating to one. Applying any *value* other than a λ-abstraction is a *type error*. We make this more precise below and come back to it in Section 1.2 on analysis.

The grammar also includes the meta-variable τ that describes the types of

x, y	::=		term variable
τ, σ	::=		type
	|	$\tau \to \tau$	function type
	|	bool	boolean type
e	::=		term
	|	**true**	true constant
	|	**false**	false constant
	|	**if** e **then** e **else** e	conditional
	|	x	term variable
	|	$\lambda x : \tau . e$	term abstraction
	|	$e\ e$	term application
v	::=		value
	|	**true**	true constant
	|	**false**	false constant
	|	$\lambda x : \tau . e$	term abstraction
Γ	::=		type context
	|	ϵ	empty context
	|	$\Gamma, x : \tau$	term binding

Figure 1.1: $\lambda_{\mathbb{B}}$ syntax

$\lambda_{\mathbb{B}}$. Each λ-abstraction contains a *type annotation* τ for the argument variable x. The denotation is that the function represented by the λ-abstraction expects a value of type τ when it is applied. We discuss types and typing contexts Γ in more detail in Section 1.1.3, which deals with static typing.

1.1.2 Semantics

We have defined the syntax of $\lambda_{\mathbb{B}}$ expressions and can now turn towards defining their meaning. There are multiple established ways to define semantics of programming language. We can coarsely classify the approaches into three different groups:

1. Operational Semantics

   Operational semantics defines the meaning of programs by specifying their execution in a state transition system. A *state transition function* or a *state transition relation* on the terms of the programming language defines the possible execution steps. The program is part of the state. For small languages the entire state might consist of only the program.

After taking a step we are left with an updated state that includes a residual program.

2. Denotational Semantics

   Denotational semantics defines the meaning of programs in terms of collection of *mathematical semantic domains* that can include numbers, sets or functions. An *interpretation function* maps program terms into these domains. This function is generally compositional in the syntax which is beneficial for modularity.

   Usually, the semantic domain has an established *formal theory*. The theorems of the domain give rise to reasoning laws for programs. Furthermore, we can also derive properties of programming languages from properties of the collection of semantic domains.

3. Axiomatic Semantics

   Instead of deriving laws for programs from their execution behaviour or denotation we can also axiomatically assume these laws. This is known as an axiomatic semantics.

   This gives us immediately the means for reasoning about programs. Furthermore, we can derive a denotational semantics for the language by constructing a model that satisfies the chosen laws and derive properties for this model or even all models.

These three approaches have different trade-offs. Denotational and axiomatic semantics immediately give us powerful mathematical tools to reason about programming languages and their programs, but for larger languages the required technicality and complexity makes it extremely difficult to even define a suitable semantics.

Operational semantics do not give us the same powerful mathematical reasoning techniques and instead impose on us the laborious task to reason about programs by observing their behaviour during execution. However, operational semantics are simpler and easier to define than more abstract denotational or axiomatic semantics. Moreover, they are much closer to actual implementations. Due to the smaller gap, operational semantics make it easier to reason about the correctness of implementations.

For our $\lambda_{\mathbb{B}}$ language we define semantics using a *small-step operational semantics*. This is also the approach used in Part II of this thesis. Part I uses denotational semantics.

Figure 1.2 gives the complete definition of the operational semantics by means of an evaluation relation. The box in the upper left corner $\boxed{e_1 \longrightarrow e_2}$

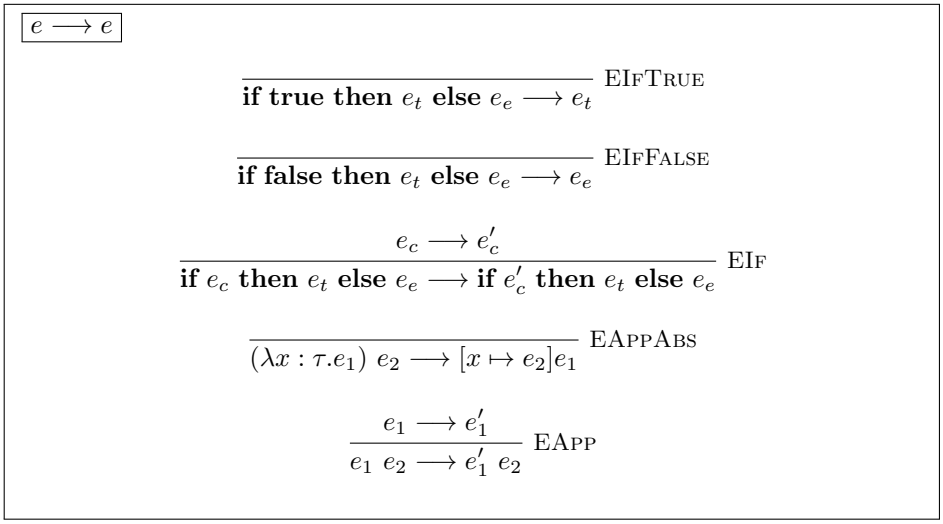


Figure 1.2: $\lambda_{\mathbb{B}}$ reduction rules

defines the shape and notation that we use for the relation. In this case it is a binary relation between two terms, which denotes that e_1 evaluates to e_2 in one step.

The remainder of the figure defines the relation using Gentzen-style inference rules [Gentzen, 1935]. In general, rules take the form

$$\frac{J_1 \qquad J_2 \qquad \ldots \qquad J_n}{J}\ \textsc{Name}$$

where Name is an optional name for the rule that allows us to refer to it. The meta-variable J stands for judgements, which in our case are usually mathematical statements in propositional or first-order logic. The judgements $J_1, \ldots, J_n$ above the horizontal line are the premises of the rule, and the judgement J below the line is the conclusion. Rule without any premises are also called *axioms*. The conclusion will always involve the relation that is being defined.

The single-step evaluation is defined using five rules that encode a *call-by-name* evaluation strategy. The two axioms EIfTrue and EIfFalse show how to reduce an **if**-expression in case the condition is either a **true** or a **false** value. The case of a condition that is not yet fully evaluated is handled by rule EIf. If the condition e_c reduces to e_c' in one step then we can conclude

the one step reduction

$$\textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e \longrightarrow \textbf{if } e_c' \textbf{ then } e_t \textbf{ else } e_e$$

The last two rules cover the evaluation of λ-terms. The rule EAppAbs handles the case where the left-hand side of an application is a λ-term $(\lambda x : \tau.e_1)$. The residual program is the the body e_1 of the function after substituting e_2 for x which we write as $[x \mapsto e_2]e_1$. Due to the call-by-name evaluation, the argument of the function does not have to be fully evaluated. If the left-hand side is not yet a λ-term, we evaluate it first similarly to EIf.

Note that this definition does not cover all possible cases. In particular the case of a λ-term in the condition of an **if**-expression

$$\textbf{if } (\lambda x : \tau.e) \textbf{ then } e_t \textbf{ else } e_e$$

and the cases of a boolean in the left-hand side of an application

$$\textbf{true } e_1 \qquad \text{or} \qquad \textbf{false } e_2$$

are not specified. Since no transition is defined and the execution is stopped without any *meaningful result*, we also say that the evaluation got stuck. In an implementation of the programming language, this corresponds to an error that can happen during the execution of a program. It's therefore also called a *(dynamic) type error*. Programmers want to detect potential problems like that early in the development cycle and, if possible, at compile time. This motivates the development of *static type systems*.

1.1.3 Typing

A *type system* is an assignment of types to expressions. Usually, not all expressions are typeable and un-typeable expressions are rejected. Also, in some languages there are expressions that can be assigned multiple, potentially incomparable types. Both the partiality and the ambiguity of types suggest a *relational* rather than a *functional* assignment. Such a relation is defined in Figure 1.3. It is a ternary relation $\boxed{\Gamma \vdash e : \tau}$ between a typing context Γ, an expression e and a type τ.

The typing relation is defined using six rules. The two rules TTrue and TFalse respectively state that the boolean constants **true** and **false** have a boolean type. The rule TIf handles the case of an **if**-expression. The three sub-expression positions contain meta-variables e_c, e_t and e_e. The premises require that the condition e_c has type boolean and the **then** and **else** branches

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \textbf{true} : \text{bool}}\ \textsc{TTrue} \qquad \frac{}{\Gamma \vdash \textbf{false} : \text{bool}}\ \textsc{TFalse}$$

$$\frac{\Gamma \vdash e_c : \text{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_e : \tau}{\Gamma \vdash \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_e : \tau}\ \textsc{TIf}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{TVar} \qquad \frac{\Gamma, y : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda y : \sigma . e) : (\sigma \to \tau)}\ \textsc{TAbs}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{TApp}$$

Figure 1.3: $\lambda_{\mathbb{B}}$ typing rules

have the same type τ. The rule then concludes that the entire **if**-expression also has type τ.

The three remaining rules deal with λ-abstractions. The typing context Γ is a list that associates term variables with types. In the case of a λ-bound variable, rule TVar looks up the corresponding type in Γ. Rule TAbs checks the body of a λ-abstraction in the context $(\Gamma, y : \sigma)$ which is the outside context Γ extended with a pair for the λ-bound variable y. The type of the λ-abstraction is the function type $(\sigma \to \tau)$ between the argument type σ and the type of the body τ. Finally, rule TApp requires that the left expression of an application has a function type that is compatible with the argument.

Example Consider the boolean negation function

$$\lambda(y : \text{bool}).\textbf{if}\ y\ \textbf{then false else true}$$

This function sends booleans to booleans and should therefore have the type bool $\to$ bool. Giving the above typing relations, we can repeatedly apply the rules to get a typing derivation for this. At each step only one possible rule applies. We can arrange the rule applications in a so called *derivation tree* that illustrates the whole derivation. Using the abbreviation $\Gamma' := \Gamma, y : \text{bool}$, we have the following tree:

$$\dfrac{\dfrac{}{\Gamma' \vdash y : \text{bool}} \quad \dfrac{}{\Gamma' \vdash \textbf{false} : \text{bool}} \quad \dfrac{}{\Gamma' \vdash \textbf{true} : \text{bool}}}{\dfrac{\Gamma' \vdash \textbf{if } y \textbf{ then false else true} : \text{bool}}{\Gamma \vdash \lambda y : \text{bool}.\textbf{if } y \textbf{ then false else true} : \text{bool} \to \text{bool}}}$$

1.2 Meta-Theoretical Analysis

As discussed before, we want to establish meta-theoretical properties that provide us with evidence for coherence of different parts of our language and to increase our confidence that a language is well-designed. This needs a rigorous and formal analysis of the defined semantics of a programming language.

In this section we showcase meta-theoretical analyses with the help of our example calculus $\lambda_{\mathbb{B}}$. In order to prove these properties we first need to understand how to reason about languages and their semantics. We therefore look at standard reasoning principles for relations like our typing and evaluation relations. We show how language properties can be expressed precisely and define two properties for our language: *determinacy of evaluation* and *type safety*, which we define below. We give a complete proof of determinacy for our language in order to demonstrate the reasoning principles. In the remainder of the thesis the type safety property plays a prominent role. Below we present an established way of proving type safety, namely through progress and preservation lemmas, and sketch the proofs.

Inductive Reasoning For our meta-theoretical analyses, we first need to establish methods for reasoning about the evaluation and typing of $\lambda_{\mathbb{B}}$. Figures 1.2 and 1.3 respectively define evaluation and typing for $\lambda_{\mathbb{B}}$. More precisely, our intention in both figures is to define the *smallest relation* that includes the presented rules. This gives rise to a *structural induction principle* for the relations. Put differently, we can induct over the shape of derivation trees (or their size or height).

As an example consider the following determinacy theorem which states that at each point there is at most one possible successor state.

Theorem 1 (Determinacy)**.** *If* $e_1 \longrightarrow e_2 \wedge e_1 \longrightarrow e_3$ *then* $e_2 = e_3$.

Proof. The proof proceeds by induction over the derivation of $e_1 \longrightarrow e_2$. For each possible case in the derivation of $e_1 \longrightarrow e_2$ we inspect the last rule that was used to derive $e_1 \longrightarrow e_3$. If the same rule was used, we can derive

the equation. All other combinations lead to a contradiction and hence the property follows.

We go through one of the inductive steps in detail and omit the others for brevity. Consider the case where $e_1 \longrightarrow e_2$ was derived using EIF. So there exist expressions $e_{11}, e'_{11}, e_{12}, e_{13}$ such that

$$e_1 = \textbf{if } e_{11} \textbf{ then } e_{12} \textbf{ else } e_{13},$$
$$e_2 = \textbf{if } e'_{11} \textbf{ then } e_{12} \textbf{ else } e_{13}$$
$$\text{and } e_{11} \longrightarrow e'_{11}.$$

Now look at the last rule that was used to derive $e_1 \longrightarrow e_3$. There are two possibilities: rule EIFTRUE or rule EIF.

1. If rule EIFTRUE was used, then we learn that $e_{11} = \textbf{true}$ and therefore $\textbf{true} \longrightarrow e'_{11}$. However this is impossible because no rule evaluates **true** further.

2. If rule EIF was used, then there exists e''_{11} such that

   $$\text{and } e_{11} \longrightarrow e''_{11}.$$

   Applying the inductive hypothesis of $e_{11} \longrightarrow e'_{11}$ to the derivation $e_{11} \longrightarrow e''_{11}$ gives us the equality $e'_{11} = e''_{11}$ from which we can derive $e_2 = e_3$.

□

Type Safety A programmer using a statically-typed language will expect certain safety guarantees from the type system when her program is executed. Intuitively, an expression of a given type will eventually evaluate to a value of that type. Usually side-conditions are implicitly assumed like for example the assumption that the computation will not diverge. In practice, there are statically-typed languages that allow non type-safe programs to be written, e.g. C and C++ are inherently unsafe. The overall convention is still that programmers write type-safe code and assume that code written by others is type-safe. In these situations type safety is a property of programs, but in this section we want to make it a property of languages, or put differently, we want that all programs of a language are type-safe.

Below we look at a formal definition of a type safety property that guarantees that the expectations of the programmer are met. Our definition will be slightly stronger than what a programmer might anticipate. The intuition

is that we disallow dynamic type errors during evaluation instead of focusing on the result of evaluation.

We first make vague concepts like *value*, or *type error* precise. Values are expressions that are canonical for the types of the language.

Definition 2 (Value)**.** *Values are the subset of expressions that are defined by the following grammar:*

v	$::=$		*term*
	$\mid$	$\textbf{\textit{true}}$	*true constant*
	$\mid$	$\textbf{\textit{false}}$	*false constant*
	$\mid$	$\lambda x : \tau.e$	*term abstraction*

By inspecting all the rules of the evaluation relation, we can see that there are no further transitions from a value. Values are thus fully evaluated[3] expressions. Such expressions are also called *normal forms*. More formally we have the following definition and lemmas.

Definition 3 (Normal Form)**.** *An expression e_1 is a* normal form *if no further execution step can be taken, i.e.,*

$$\forall e_2.\neg(e_1 \longrightarrow e_2)$$

Lemma 4 (Values are Normal)**.** *If an expression e is a value, then it's also a normal form.*

Proof. By inspecting the evaluation rules for each value form. □

We can thus translate our intuitive understanding of type safety into a theorem:

Theorem 5 (Type Safety (First Attempt))**.** *Let e_1 be an expression of type τ, i.e. $\cdot \vdash e_1 : \tau$. If e_1 evaluates in one or more steps to a value e_2, then e_2 also has type τ:*

$$\forall e_2.(e_1 \longrightarrow^* e_2 \wedge e_2 \textit{ is a value}) \Rightarrow \cdot \vdash e_2 : \tau.$$

This definition however is problematic to work with directly. First, we cannot prove it directly by induction over e_1 or over the typing derivation $\cdot \vdash e_1 : \tau$, because in the case of an evaluation step with rule EAppAbs we are

[3]Fully evaluated with respect to the given semantics. Redexes may still appear under λ-abstractions which are *suspended*. There are also semantics that allow evaluation under λ-abstractions.

given a reduced term that is not a sub-term and hence we have no induction hypothesis available. Second, it does not express strong coherence between typing and evaluation. Consider for instance the language that we get, if we remove the evaluation rule EIfTrue. Then the expression

$$\textbf{if true then } e_t \textbf{ else } e_e$$

is a normal form but not a value. However, it is different than for example the expression

$$\textbf{if } (\lambda x : \tau.e) \textbf{ then } e_t \textbf{ else } e_e$$

which is also a normal from but not a value. But intuitively, evaluations that stopped should either be values or type errors.

With this understanding we can reformulate our type safety theorem. Type safe languages rule out type errors; consequently, normal forms should already be values.

Theorem 6 (Type Safety)**.** *Let* $\cdot \vdash e_1 : \tau$*. If* e_1 *evaluates to a normal form* e_2 *then* e_2 *is a value of type* τ*:*

$$(e_1 \longrightarrow^* e_2 \wedge e_2 \textit{ is normal}) \Rightarrow (e_2 \textit{ is a value} \wedge \cdot \vdash e_2 : \tau).$$

Note that this definition does not require the evaluation to terminate. A program that runs forever without getting stuck is also considered type-safe.

Proving this property directly is difficult as well. One established and popular way is to reduce it to two simpler properties, namely *progress* and *preservation*. Progress expresses that we can always take a step as long as we do not reach a value.

Lemma 7 (Progress)**.** *Let* $\cdot \vdash e_1 : \tau$*. Either* e_1 *is a value or we can take another step, i.e.*

$$\exists e_2.e_1 \longrightarrow e_2.$$

Proof (Sketch). By induction over the typing derivation $\cdot \vdash e_1 : \tau$. If we can take an evaluation step in a sub-term we can use rule EIf rule EApp. Otherwise, we learn that we have a value in an evaluation position. By inspecting the possible values for a given type we can take a step with one of the rules EIfTrue, EIfFalse or EAppAbs. □

Lemma 8 (Preservation)**.** *If* $\Gamma \vdash e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $\Gamma \vdash e_2 : \tau$*.*

Proof (Sketch). By induction over the typing derivation $\Gamma \vdash e_1 : \tau$ and inspection of the last rule to derive $e_1 \longrightarrow e_2$. The only interesting case is EAppAbs which follows from an additional lemma that states that typing is preserved by well-typed substitutions. In all other cases the property follows immediately from an induction hypothesis or by application of a single rule and the induction hypotheses. □

Proof of Theorem 6. By induction over the number of evaluation steps in $e_1 \longrightarrow^* e_2$ and using the progress and preservation lemmas. □

1.3 Mechanization

Meta-theoretical proofs are long and require the management of many details. Consequently, these proofs are prone to error and it is easy for human verifiers and reviewers to overlook mistakes. This is aggravated by the fact, that properties tend to fail in subtle edge cases. For instance, the unsoundness example of Amin and Tate [2016] for Java involves the interplay between constrained wildcards of generics and null references. Hence, there is a real danger that an overlooked detail or a falsely assumed assumption leads to both invalid proofs and invalid results.

Theorem provers In order to gain more confidence in the correctness of meta-theory proofs, the field of programming language theory has started to adopt *mechanization* as a new methodology: writing mathematical theorems in a formal language and verify correctness of their proofs mechanically by *automated theorem provers*. Theorem provers may find proofs for theorems fully automatically or require a human user to input the proof or proof hints. In the latter case, the system is also called a *proof assistant*. In both cases, every reasoning step of the proof is verified to be valid in the system's underlying logic. This has the benefit that gaps in the proof are ruled out and unproven assumptions need to be explicitly documented. Therefore, the use of theorem provers greatly increases the confidence in the validity of (meta-theory) proofs.

Among programming language researchers proof assistants are more popular, because they provide richer logics. Systems like Abella [Gacek, 2008], Agda [Norell, 2007], Beluga [Pientka and Dunfield, 2010], Coq [Coquand et al., 1984], Isabelle/HOL [Nipkow et al., 2002] and Twelf [Pfenning and Schürmann, 1999] have been used to check various meta-theoretic proofs. The developments of this thesis have been done in the context of the Coq proof assistant which

offers a competitive degree of automation and is one of the most widely used proof assistants for meta-theory.

Benefits To illustrate the benefits of mechanizations, consider the elaborate bug hunting study of Yang et al. [2011] using their randomized test-case generator Csmith. During their study they found and reported over 325 bugs in 11 open source and commercial C compilers. One of the compilers is CompCert [Leroy, 2009], a C compiler implemented in the Coq proof assistant [Coquand et al., 1984] and proven to be correct with respect to a formal specification of the C programming language. Yang et al. [2011] write the following about CompCert:

> The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Costs Unfortunately, mechanization does not address the large effort of proving properties, but rather aggravates it as every little detail has to be spelled out. The size depends largely on the language and the properties that are proved. Our example language $\lambda_{\mathbb{B}}$ can be specified in about 94 lines[4] of Coq code and type safety including all necessary lemmas can be proven in 55 lines of code. However, this is not representative since $\lambda_{\mathbb{B}}$ is a bare-bones calculus. It is not uncommon to see developments with few hundreds to thousands of lines of language specifications and tens of thousands of lines of meta-theory proofs [Leroy, 2009; Zhao et al., 2010].

1.4 Reusability

Despite the benefits, neither formal specification of programming languages nor rigorously mechanised meta-theory proofs are widespread practice. One of the main obstacles are the large development costs. This thesis aims to help

[4]Including definitions of capture-avoiding substitutions that we omitted for brevity.

spur further adoption of formal mechanised meta-theory by promoting *reuse* as a method to lower the mechanization effort.

Reuse is a common approach in software engineering to reduce development cost and increase both quality and reliability. The idea is to identify functionality or patterns that different software systems have in common and implement them only once in a manner that can be shared by the different software systems and reused in the development of new systems. We apply the same idea to programming language meta-theory, identifying repeated functionality and patterns, and implementing them only once in a way that can be used across proofs for different languages.

Unfortunately, the current practice to achieve reuse is to copy an existing formalization, *change the existing definitions* manually, integrate new features and to subsequently *patch up the proofs* to cater for the changes. This unprincipled approach to reuse leaves much to be desired. First, editing and patching the existing definitions breaks *abstraction*, a core principle of computer science. Ideally, we would like to reuse existing code via an interface that provides functionality (for programming) and properties (for reasoning). Second, this approach does not encourage *isolation* of new features from existing ones, which hinders backporting improvements to the existing formalization.

Our goal is to replace the current practice with principled ways to achieve reusability. More specifically, this thesis is examining two different means of reuse: 1. Through *modularity* and 2. through *genericity*.

Modularity Programming languages, just like regular software systems, can be described by the functionality or features that they provide. The meta-theoretic development of our example language $\lambda_{\mathbb{B}}$ consists of different functionality: the syntax, semantics and its type safety proof. It easy to reuse the syntax and semantics and prove other kinds of properties, e.g. termination, or, to reuse the syntax and switch out the semantics and prove type safety for the new semantics.

Furthermore, $\lambda_{\mathbb{B}}$ has two easily distinguishable features: λ-terms and boolean expressions. Many programming languages and their calculi feature either or both of these language constructs among many others. This commonality hints at another opportunity for reuse. However, achieving this kind of reuse is challenging (cf. Section 2.1).

Ideally, each feature could be developed in complete isolation and modularly combined into a full language consisting of a specific set of features. The reality, however, is more complicated. Features may have dependencies between them or may interact with each other even though they seem to be

orthogonal.

This thesis investigates the modularization of meta-theory proofs along feature boundaries and specifically looks at the reduction of interaction between side-effecting language features.

Genericity Names are found in almost every high-level programming language to refer to classes, methods, types, functions, function parameters, etc. In case a name is *substitutable* we call it a *variable*. A language may have multiple kinds of variables, for example, term and type variables. The operational semantics of languages with variables often implement reduction of language constructs by means of *substitution*.

Variable substitution is an operation that is not specific to a particular language or language feature but is common to any language that uses variable binding. The implementation of substitution follows a standard recipe that can be applied for any specific language. This hints at a different way of achieving reuse: by implementing substitution *once* generically and specializing this generic implementation to any given concrete language.

While other kinds of *generic functionality* are commonly found in programming language meta-theory, e.g. term equality or (first-order) unification [Van Noort et al., 2010], the generic functionality considered in this thesis is substitution. Substitution is interesting functionality for reuse because the need for substitutions arises in nearly all meta-theory proofs for languages with variable binding. Furthermore, many theorems need a large amount of lemmas about substitutions. Discharging them automatically, e.g. through reusable generic implementations, can save a lot of development effort.

1.5 Overview

This thesis is split into two parts, corresponding to the two means of achieving reuse with the discussed focus.

Part I: Modularity Chapter 2 presents the necessary background for this part. While modular development of software is a well-studied topic in computer science, modular composition of proofs is not as well-studied. Chapter 3 develops one approach to modularized algebraic dataypes and modularized induction proofs for them. This chapter contains the material from

> Keuchel, S. and Schrijvers, T. (2013). Generic Datatypes à la Carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, WGP 13, pages 13-24. ACM.

Side-effecting language features exhibit a lot of interaction, e.g. in operational semantics, which hinders modularization. This is a problem that is tackled in Chapter 4 by developing a monadic denotational semantics for features with side-effects that can be modularized. This chapter is based on the publication

> Delaware, B., Keuchel, S., Schrijvers, T., and Oliveira, B. C. d. S. (2013). Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 319-330. ACM.

Part II: Genericity This part deals with the development of a generic substitution operation and associated theorems. The solution put forward is a framework consisting of a specification language KNOT for abstract syntax with variable binding and the tool NEEDLE that compiles KNOT-specifications to Coq code which includes substitution operators and proofs about substitutions. Chapters 6 and 7 present the syntax and semantics of KNOT respectively, while Chapter 8 discusses the elaboration and code generation underlying NEEDLE.

Part II is based on

> Keuchel, S., Weirich, S., and Schrijvers, T. (2016). Needle & Knot: Binder Boilerplate Tied Up. In *Programming Languages and Systems: 25th European Symposium on Programming*, ESOP '16, pages 419–445. Springer.

which deals with substitutions at the term level and

> Keuchel, S., Schrijvers, T., and Weirich, S. (2016). Needle & Knot: Boilerplate Bound Tighter. Unpublished draft.

which extends the framework further to include predicates on terms expressed as relations.

Not included Related to the content of this thesis, but not included, are the articles

> Keuchel, S. and Schrijvers, T. (2012). Modular Monadic Reasoning, a (Co-)Routine. Presented at *the 24th Symposium on Implementation and Application of Functional Languages*, IFL '12.

which develops initial ideas to modular reasoning about side-effecting components, and

> Keuchel, S. and Schrijvers, T. (2015). InBound: Simple yet powerful Specification of Syntax with Binders. Unpublished draft.

which aims to develop a richer specification language of syntax with binding than KNOT, including binding constructs of realistic programming languages that KNOT does not support.

Furthermore, the experiences gained from the work on NEEDLE & KNOT are used to tackle the substitution boilerplate of

> Devriese, D., Patrignani, M., Piessens, F., and Keuchel, S. (2017). Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science*, Volume 13, Issue 4.

Part I

Modularity

Chapter 2

Background

Formal mechanization of programming language meta-theory is a big endeavor due its unwieldy size, complexity and attention to detail. Therefore reuse is crucial and indeed, the POPLMARK challenge [Aydemir et al., 2005] identifies lack of *component reuse* as one of several key obstacles of large-scale mechanizations.

It is therefore desirable to apply established and *principled software engineering methods* for reusability to programming language mechanization. The objective of these software engineering methods is *modularity*: to build new software systems entirely from reusable components, that have been written independently and can (potentially) be reused in many different configurations for different applications.

A stumbling block for applying the modularity principle to programming language formalization is that traditional inductive definitions and proofs are closed to extension. It is therefore necessary to develop modular reasoning principles first; in fact, *modular induction principles*, because we want to stay as close to established proof techniques as possible. Opening induction definitions for extensibility is a manifestation of the *expression problem* [Wadler, 1998].

This chapter covers background information on modular reasoning that forms the basis for Chapter 3 and 4. In particular, we summarize techniques that are used in the *Meta-Theory à la Carte* (MTC) framework, an existing solution for modular reasoning about programming languages in Coq, on which we build. However, we limit ourselves to parts that are relevant to Chapters

```
data ArithExp
   = Lit Int
   | Add ArithExp ArithExp
eval :: ArithExp → Int
eval (Lit i)       = i
eval (Add e1 e2) = eval e1 + eval e2
```

Figure 2.1: Evaluation of arithmetic expressions (Haskell)

```
interface ArithExp {
  int eval ();
}
class Lit implements ArithExp {
  public int lit;
  public int eval () {return lit; }
}
class Add implements ArithExp {
  public ArithExp e1, e2;
  public int eval () {return e1.eval () + e2.eval (); }
}
```

Figure 2.2: Evaluation of arithmetic expressions (Java)

3 and 4 and refer the reader to the original paper [Delaware et al., 2013] for full details.

2.1 Expression Problem

Consider the Haskell program for the evaluation of simple arithmetic expressions in Figure 2.1. We have a datatype *ArithExp*, representing arithmetic expressions with constructors for integer literals and addition, and an evaluation function $eval :: ArithExp \rightarrow Int$ that evaluates an expression to an integer value.

Figure 2.2 shows an equivalent Java program. The *ArithExp* interface

```
data ArithExp
   = Lit Int
   | Add ArithExp ArithExp
   | Mul ArithExp ArithExp
eval :: ArithExp → Int
eval (Lit i)       = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
print :: ArithExp → String
print (Lit i)       = show i
print (Add e1 e2) = "(" ++ print e1 ++ "+" ++ print e2 ++ ")"
print (Mul e1 e2) = "(" ++ print e1 ++ "*" ++ print e2 ++ ")"
```

Figure 2.3: Extended arithmetic expressions (Haskell)

contains an *int eval* () method. The two cases of a literal and an addition are handled by the two classes *Lit* and *Add* that implement *ArithExp*.

We can extend these programs along two dimensions:

1. Adding a new case, e.g. a constructor for multiplication.

2. Adding a new operation, e.g. converting an expression to a string.

In Haskell, performing the second extension in our example is easy: we add one more function to the program

```
print :: ArithExp → String
print (Lit i)       = show i
print (Add e1 e2) = "(" ++ print e1 ++ "+" ++ print e2 ++ ")"
```

However, covering a new case inevitably requires modifying existing code: it has to be added to the *ArithExp* datatype declaration and, for totality, also to existing functions. Figure 2.3 shows the code with both extensions.

In Java the situation is reversed. The multiplication case can easily be added by creating a new class *Mul* that implements *Exp*.

```
class Mul implements ArithExp {
  public ArithExp e1, e2;
```

```
interface ArithExp {
  int eval ();
  String print ();
}
class Lit implements ArithExp {
  public int lit;
  public int eval () {return lit; }
  public String print () {return String.valueOf (lit); }
}
class Add implements ArithExp {
  public ArithExp e1, e2;
  public int eval () {return e1.eval () + e2.eval (); }
  public String print () {
    return e1.print ().concat ("+").concat (e2.print ());
  }
}
class Mul implements ArithExp {
  public ArithExp e1, e2;
  public int eval () {return e1.eval () * e2.eval (); }
  public String print () {
    return e1.print ().concat ("*").concat (e2.print ());
  }
}
```

Figure 2.4: Extended arithmetic expressions (Java)

```
  public int eval () {return e1.eval () * e2.eval (); }
}
```

However, the conversion to a *String* inevitably requires editing the existing code and adding a new method to the *ArithExp* interface and existing implementations of that interface.[1] Figure 2.4 shows the code with both extensions.

Performing such extensions in both dimensions simultaneously and modularly, i.e. without changing or recompiling the existing code, and keeping the code type-safe was coined as *the expression problem* by Wadler [1998]. Solutions to the expression problem exist in multiple languages: Wadler [1998]

[1] We disregard the *toString* () method that is part of the base class *Object*.

$$\begin{array}{l}\textbf{data}\ \mathit{Exp} = \mathit{Lit}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \\ \qquad\qquad \mid \mathit{BLit}\ \mathit{Bool} \mid \mathit{If}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Exp}\end{array}$$

$$\begin{array}{l}\textbf{data}\ \mathit{Arith}_F\ \mathit{exp} = \mathit{Lit}_F\ \mathit{Int} \mid \mathit{Add}_F\ \mathit{exp}\ \mathit{exp} \\ \textbf{data}\ \mathit{Logic}_F\ \mathit{exp} = \mathit{BLit}_F\ \mathit{Bool} \mid \mathit{If}_F\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\end{array}$$

Figure 2.5: Arithmetic and logical expressions

presents a solution in Java using Generics and the *Datatypes à la Carte* (DTC) approach [Swierstra, 2008] is a well-known solution in the Haskell programming language. In both of these solutions, modularity has to be anticipated and catered for from the beginning however. Indeed, we cannot reuse the datatype declaration and interface declarations from this section, but have to use ones that account for modular extensions. We will call a Haskell datatype that can be modularly extended a *modular datatype* and use the term *modular function* for modularly extensible functions that are defined on modular datatypes.

Our goal to modularly engineer programming language meta-theory adds a third dimension to the expression problem: *modular proofs* of statements about modular functions on modular datatypes. In the remainder of this chapter we first present the DTC approach (Section 2.2) and then discuss stumbling and building blocks to extend the approach to support modular reasoning (Sections 2.3, 2.4 and 2.5).

2.2 Datatypes à la Carte

This section reviews the core ideas behind *Datatypes à la Carte* (DTC) [Swierstra, 2008], a well-known Haskell solution to the expression problem, and presents the infrastructure for writing modular functions over modular datatypes.

2.2.1 Fixed-points

The main idea behind DTC is to open the recursion of datatypes and model the fixed point explicitly. Consider the monolithic datatype *Exp* for simple arithmetic and logical expressions in Figure 2.5 (top). Abstracting over the recursive positions of *Exp* yields a *signature functor* that we can then split up

```
data Fix_D f       = In_D {out_D :: f (Fix_D f)}
data (⊕)   f g a = Inl (f a) | Inr (g a)
```

Figure 2.6: Datatypes à la Carte fixed-point

into functors $Arith_F$ and $Logic_F$ – shown in Figure 2.5 (bottom) – to capture the signature of features in isolation.

The type-level fixed-point combinator Fix_D in Figure 2.6 creates a recursive datatype from a signature. For example $Arith_D$ is a type that features only arithmetic expressions.

```
type Arith_D = Fix_D Arith_F
```

Different features can be combined modularly by taking the coproduct ($\oplus$) of the signatures before taking the fixed point. For example, taking the fixed-point of the coproduct of $Arith_F$ and $Logic_F$

```
type Exp_D = Fix_D (Arith_F ⊕ Logic_F)
```

essentially[2] yields a datatype that is isomorphic to the monolithic datatype *Exp* from Figure 2.5 (top).

2.2.2 Automated Injections

Combining signatures makes writing expressions difficult. For example the arithmetic expression $3 + 4$ is represented as the term

```
ex1 :: Fix_D (Arith_F ⊕ Logic_F)
ex1 = In_D (Inl (Add_F
                 (In_D (Inl (Lit_F 3)))
                 (In_D (Inl (Lit_F 4)))))
```

Writing such expressions manually is too cumbersome and unreadable. Moreover, if we extend the datatype with a new signature, other injections are needed.

To facilitate writing expressions and make reuse possible we use the subfunctor $f \prec: g$ relation shown in Figure 2.7 (top). The member function

[2]Which due to laziness in Haskell means modulo non-termination.

```
class f ≺: g where
  inj     :: f a → g a
  prj     :: g a → Maybe (f a)
  inj_prj :: ∀a (ga :: g a) (fa :: f a).
               prj ga = Just fa → ga = inj fa
  prj_inj :: ∀a (fa :: f a).
               prj (inj fa) = Just fa
inject :: (f ≺: g) ⇒ f (Fix_D g) → Fix_D g
inject x = In_D (inj x)
project :: (f ≺: g) ⇒ Fix_D g → Maybe (f (Fix_D g))
project x = prj (out_D x)
```

```
instance (f ≺: f) where
  inj = id
instance (f ≺: g) ⇒ (f ≺: (g ⊕ h)) where
  inj = Inl ∘ inj
instance (f ≺: h) ⇒ (f ≺: (g ⊕ h)) where
  inj = Inr ∘ inj
```

Figure 2.7: Sub-functor relation

inj injects the sub-functor f into the super-functor g. In our case we need injections of functors into coproducts which are automated using type class machinery.[3] The *prj* member function is a partial inverse of *inj*. With it we can test if a specific sub-functor was used to build the top layer of a value. This operation fails if another sub-functor was used. The type class also includes the laws *inj_prj* and *prj_inj* that witness the partial inversion.[4]

The *inject* function is a variation of *inj* that additionally applies the constructor of the fixed-point type Fix_D. Using the sub-functor relation we can define smart constructors for arithmetic expressions

```
lit :: (Arith_F ≺: expf) ⇒ Int → Fix_D expf
lit i = inject (Lit_F i)
```

[3] Coq's type-class mechanism performs backtracking. These instances do not properly work in Haskell. See [Swierstra, 2008] for a partial solution.

[4] Using a hypothetical dependently-typed Haskell syntax.

```
data IntValue_F  val = VInt  Int
data BoolValue_F  val = VBool Bool
data StuckValue_F val = VStuck
vint :: (IntValue_F ≺: valf) ⇒ Int → Fix_D valf
vint i = inject (VInt i)
vbool :: (BoolValue_F ≺: valf) ⇒ Bool → Fix_D valf
vbool b = inject (VBool b)
vstuck :: (StuckValue_F ≺: valf) ⇒ Fix_D valf
vstuck = inject VStuck
```

Figure 2.8: Modular value datatype

```
add :: (Arith_F ≺: expf) ⇒ Fix_D expf → Fix_D expf → Fix_D expf
add a b = inject (Add_F a b)
```

that construct terms of any abstract super-functor *expf* of $Arith_F$. This is essential for modularity and reuse. We can define terms using the smart-constructors, but constructing a value of a specific fixed-point datatype is delayed. With these smart constructors the above example term becomes

```
ex1' :: (Arith_F ≺: expf) ⇒ Fix_D expf
ex1' = lit 3 `add` lit 4
```

The *project* function is a variation of *prj* that strips the constructor of the fixed-point type Fix_D. Similarly to injections, we can automate projections for coproducts by adding corresponding definitions to the instances above.

2.2.3 Semantic Functions

In this section we define evaluation for arithmetic and boolean expressions modularly. We use another modular datatype to represent values. Its signatures and smart-constructors are given in Figure 2.8. The signature $StuckValue_F$ represents a sentinel value to signal type errors during evaluation.

If f is a functor, we can fold over any value of type $Fix_D\ f$ as follows:

```
type Algebra f a = f a → a
fold_D :: Functor f ⇒ Algebra f a → Fix_D f → a
fold_D f (In_D x) = f (fmap (fold_D f) x)
```

```
class FAlgebra name f a where
  f_algebra :: name → Algebra f a
algebraPlus :: Algebra f a → Algebra g a → Algebra (f ⊕ g) a
algebraPlus f g (Inl a) = f a
algebraPlus f g (Inr a) = g a
instance (FAlgebra name f a, FAlgebra name g a) ⇒
         FAlgebra name (f ⊕ g) a where
  f_algebra name = algebraPlus (f_algebra name) (f_algebra name)
```

Figure 2.9: Function algebra infrastructure

An *algebra* specifies one step of recursion that turns a value of type $(f\ a)$ into the desired result type a. The *fold* uniformly applies this operation to an entire term. All semantic functions over a modular datatype are written as folds of an algebra.

Using type classes, we can define and assemble algebras in a modular fashion. The class *FAlgebra* in Figure 2.9 carries an algebra for a functor f and carrier type a. It is additionally indexed over a parameter *name* to allow definitions of distinct functions with the same carrier. For instance, functions for calculating the size and the height of a term can both be defined using *Int* as the carrier.

We use the name *Eval* to refer to the evaluation algebra.

```
data Eval = Eval
```

The evaluation algebras are parameterized over an abstract super-functor *valf* for values. In case of $Arith_F$ we require that integral values are part of *valf* and for $Logic_F$ we require that boolean values are part of *valf*.

In the case of an Add_F in the evaluation algebra for arithmetic expressions we need to project the results of the recursive calls to test whether integral values were produced. Otherwise a type error occurs and the *stuck* value is returned.

```
instance (IntValue_F ≺: valf, StuckValue_F ≺: valf) ⇒
         FAlgebra Eval Arith_F (Fix_D valf) where
  f_algebra Eval (Lit_F i)     = vint i
  f_algebra Eval (Add_F a b) = case (project a, project b) of
```

$$
\begin{array}{ll}
(\mathit{Just}\ (\mathit{VInt}\ a), \mathit{Just}\ (\mathit{VInt}\ b)) & \to \mathit{vint}\ (a + b) \\
\_ & \to \mathit{vstuck}
\end{array}
$$

Similarly, we have to test the result of the recursive call of the condition of an If_F term for boolean values.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{BoolValue}_F \prec: \mathit{valf}, \mathit{StuckValue}_F \prec: \mathit{valf}) \Rightarrow \\
\qquad \mathit{FAlgebra}\ \mathit{Eval}\ \mathit{Logic}_F\ (\mathit{Fix}_D\ \mathit{valf})\ \textbf{where} \\
\quad f\_\mathit{algebra}\ \mathit{Eval}\ (\mathit{BLit}_F\ b)\ = \mathit{vbool}\ b \\
\quad f\_\mathit{algebra}\ \mathit{Eval}\ (\mathit{If}_F\ c\ t\ e) = \textbf{case}\ \mathit{project}\ c\ \textbf{of} \\
\qquad \mathit{Just}\ (\mathit{VBool}\ b) \to \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ e \\
\qquad \_ \qquad\qquad\ \to \mathit{vstuck}
\end{array}
$$

Function algebras for different signatures can be combined to get an algebra for their coproduct. The necessary instance declaration is also given in Figure 2.9. Finally, we can define an evaluation function for terms given an *FAlgebra* instance for *Eval*.

$$
\begin{array}{l}
[\![\cdot]\!] :: (\mathit{Functor}\ \mathit{expf}, \mathit{FAlgebra}\ \mathit{Eval}\ \mathit{expf}\ (\mathit{Fix}_D\ \mathit{valf})) \Rightarrow \\
\quad \mathit{Fix}_D\ \mathit{expf} \to \mathit{Fix}_D\ \mathit{valf} \\
[\![\cdot]\!] = \mathit{fold}_D\ (f\_\mathit{algebra}\ \mathit{Eval})
\end{array}
$$

2.3 Reasoning à la Carte

Our goal is to extend the DTC's approach from *modular programming* to include *modular reasoning*. In this Section, we give a brief overview of reasoning in proof-assistants in general before moving towards modular reasoning in the next Sections. Moreover, we also discuss restrictions of the proof-assistant settings, which are necessary for logical consistency, but which prevents us from porting the Haskell definitions to a proof-assistant directly.

2.3.1 Propositions as Types

Researchers in logic and computer science have discovered parallels between logics and type systems for λ-calculi. For example, the connectives of propositional logic correspond to a simply-typed lambda calculus with some base type, disjoint sums and cartesian products [Curry, 1934]. Hence types of the simply-typed lambda calculus can also be interpreted as propositions in propositional

Implication	$A \Rightarrow B$	Function	$A \to B$
Disjunction	$A \vee B$	Disjoint sum	$A + B$
Conjunction	$A \wedge B$	Product	$A \times B$
Universal	$\forall x \in A.B[x]$	Dependent product	$\Pi_{(x:A)}B(x)$
Existential	$\exists x \in A.B[x]$	Dependent sum	$\Sigma_{(x:A)}B(x)$

Figure 2.10: Correspondences for propositional and predicate logic

logic. Figure 2.10 shows several logical connectives and their type counterparts. Moreover, a program of a particular type encodes a constructive proof of the corresponding proposition. Hence proving is just programming.

This correspondence is not limited to propositional logic and simple types, but has been observed for a variety of logics: propositional, predicate, second-order, intuitionistic, classical, modal, and linear logics. It is also known as the Curry-Howard correspondence or the propositions-as-types interpretation. [Wadler, 2015] gives a nice write-up and a historic account.

This correspondence serves as the basis for many type-theory based proof-assistant like Agda, Coq, NuPRL and Twelf. These systems support in particular dependent-types which correspond to universal and existential quantifiers in predicate logic (see Figure 2.10, bottom) that allows us to implement complex mathematical properties.

2.3.2 Induction Principles

Another feature commonly found in proof-assistants are inductive datatype definitions and reasoning about values of inductive types via recursion. As an example, consider a definition of natural numbers and addition:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}\\
&\mathit{plus} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{plus}\ \mathit{Zero}\ \ \ \ \ \ n = n\\
&\mathit{plus}\ (\mathit{Succ}\ m)\ n = \mathit{Succ}\ (\mathit{plus}\ m\ n)
\end{aligned}
$$

To prove a simple proposition like the right-neutrality of $Zero$ we can write a function that follows the recursive structure of the *plus* function:

$$
\begin{aligned}
&\mathit{plusZero} :: \forall m :: \mathit{Nat}.\mathit{plus}\ m\ \mathit{Zero} = m\\
&\mathit{plusZero}\ \mathit{Zero}\ \ \ \ \ \ = \mathit{Refl}\\
&\mathit{plusZero}\ (\mathit{Succ}\ m) = \mathit{cong}\ \mathit{Succ}\ (\mathit{plusZero}\ m)
\end{aligned}
$$

In the same way we implement functions using recursion schemes in programming, we can implement proofs using similar schemes. These schemes are called *induction schemes* or *induction principles*. For example, for the natural numbers we can implement the following induction principle:

$$
\begin{aligned}
&\mathit{indNat} :: \\
&\quad \forall P :: \mathit{Nat} \to \mathit{Prop}. \\
&\qquad P\ \mathit{Zero} \to \\
&\qquad (\forall m :: \mathit{Nat}.P\ m \to P\ (\mathit{Succ}\ m)) \to \\
&\qquad \forall m :: \mathit{Nat}.P\ m \\
&\mathit{indNat}\ P\ \mathit{pzero}\ \mathit{psucc} = \mathit{go} \\
&\quad \textbf{where} \\
&\qquad \mathit{go} :: \forall m :: \mathit{Nat}.P\ m \\
&\qquad \mathit{go}\ \mathit{Zero}\ \ \ \ \ \ = \mathit{pzero} \\
&\qquad \mathit{go}\ (\mathit{Succ}\ m) = \mathit{psucc}\ (\mathit{go}\ m)
\end{aligned}
$$

Notice the similarity to the natural number fold. In fact, *indNat* is a dependent version of the fold. The second and third argument, named *pzero* and *psucc* after the corresponding constructors, are called the *proof algebra* analogously to the term *algebra* used for folds. Analogously to re-implementing *plus* using folds, we can reimplement *plusZero* using the induction principle:

$$
\begin{aligned}
&\mathit{plusZero2} :: \forall m :: \mathit{Nat}.\mathit{plus}\ m\ \mathit{Zero} = m \\
&\mathit{plusZero2} = \mathit{indNat}\ (\lambda m \to \mathit{plus}\ m\ \mathit{Zero} = m)\ \mathit{Refl}\ (\mathit{cong}\ \mathit{Succ})
\end{aligned}
$$

Induction principles give us a way to open the recursive structure of proofs. Hence, we can implement modular induction proofs in the same style as we implement modular functions. The latter are expressed in terms of modular algebras and a generic fold operator. Similarly, the former are expressed in terms of modular proof algebras and a generic induction principle.

2.3.3 Strict Positivity

Unfortunately, we cannot directly translate the generic definition of the *Datatypes à la Carte* approach of Section 2.2, namely the type-level fixpoint, to a proof-assistant. These assistants commonly require all datatype definitions to be *strictly-positive* so that all datatypes denote proper inductive definitions. Lifting this restriction, i.e. allowing arbitrary non strictly-positive recursive datatypes, renders the theory of the proof-assistant inconsistent [Chlipala, 2013].

We define *strictly positive types* (SPT) by using the following generative grammar [Abbott et al., 2005]:

$$\tau ::= X \mid 0 \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid K \to \tau \mid \mu X.\tau$$

where X ranges over type variables and K ranges over constant types, i.e. an SPT with no free type variables. The constants 0 and 1 represent the empty and unit types, the operators $+$, $\times$, $\to$ and μ represent coproduct, cartesian product, exponentiation and least fixed point construction.

For $Fix_D\ f$ from Section 2.2 to be strictly positive means that the argument functor f has to be strictly-positive, i.e. it corresponds to a term built with the above grammar with one free type variable.

As a counter example, inlining the non-strictly positive functor $X \mapsto (X \to Int) \to Int$ into Fix_D yields the datatype declaration

$$\textbf{data}\ NSP = NSP\ ((NSP \to Int) \to Int)$$

This is a valid Haskell declaration, but it does not satisfy the positivity requirements and is hence rejected by Coq. While Coq can automatically determine the positivity for any concrete functor by inspecting its definition, it cannot do so for an abstract functor like the one that appears in the definition of Fix_D. Hence, Coq conservatively rejects Fix_D.

Of course, we have no intention of using non-strictly positive functors for our application and would like to provide the evidence of strict-positivity to the fixpoint type constructor. Mini-Agda [Abel, 2010] for example allows programmers to annotate strictly-positive and negative positions of type constructors. Unfortunately, Agda and Coq do not provide us with this possibility and a different approach is needed to define type-level fixed-points and fold operators.

2.4 Church Encodings

The Meta-Theory à la Carte (MTC) [Delaware et al., 2013] framework's solution to define type-level fixed-points in a proof-assistant setting is to use Church encodings, or Böhm-Berarducci encodings to be precise [Böhm and Berarducci, 1985], to encode strictly-positive algebraic datatypes.

2.4.1 Encoding Algebraic Datatypes

The untyped λ-calclus only provides functions as primitives. Yet, this is no limitation as they can be used to encode other datatypes. This technique was first used by Alonzo Church and is hence named Church encoding. For instance, the Church encoding of natural numbers is known as Church numerals. The idea is that the Church numeral c_n for the natural number n applies

$$
\begin{aligned}
&\textbf{type } \mathit{Fix}_C\ f = \forall a.\mathit{Algebra}\ f\ a \to a\\
&\mathit{fold}_C :: \mathit{Algebra}\ f\ a \to \mathit{Fix}_C\ f \to a\\
&\mathit{fold}_C\ \mathit{alg}\ x = x\ \mathit{alg}\\
&\mathit{in}_C :: \forall f.\mathit{Functor}\ f \Rightarrow f\ (\mathit{Fix}_C\ f) \to \mathit{Fix}_C\ f\\
&\mathit{in}_C\ x = \lambda \mathit{alg} \to \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}_C\ \mathit{alg})\ x)\\
&\mathit{out}_C :: \forall f.\mathit{Functor}\ f \Rightarrow \mathit{Fix}_C\ f \to f\ (\mathit{Fix}_C\ f)\\
&\mathit{out}_C = \mathit{fold}_C\ (\mathit{fmap}\ \mathit{in}_C)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inject} :: (\mathit{Functor}\ g, f \prec: g) \Rightarrow f\ (\mathit{Fix}_C\ g) \to \mathit{Fix}_C\ g\\
&\mathit{inject}\ x = \mathit{in}_C\ (\mathit{inj}\ x)\\
&\mathit{project} :: (\mathit{Functor}\ g, f \prec: g) \Rightarrow \mathit{Fix}_C\ g \to \mathit{Maybe}\ (f\ (\mathit{Fix}_C\ g))\\
&\mathit{project}\ x = \mathit{prj}\ (\mathit{out}_C\ x)
\end{aligned}
$$

Figure 2.11: Fixed-points and fold using Church encodings

a function s n-times to a value z similarly to how we get n by taking n-times the successor of zero. We can construct the Church numeral for any concretely given natural number:

$$
\begin{aligned}
&c_0 = \lambda s.\lambda z.z\\
&c_1 = \lambda s.\lambda z.s\ z\\
&c_2 = \lambda s.\lambda z.s\ (s\ z)\\
&\ldots
\end{aligned}
$$

In other words, the n-th Church numeral corresponds to the fold over natural numbers instantiated for the number n. In fact, typing the above combinators in Haskell yields the familiar type $c_n :: \forall a.(a \to a) \to a \to a$. Böhm and Berarducci [1985] proved that such an encoding is not limited to simple datatypes like the naturals, but that all strictly-positive (and parameterized) datatypes can be encoded in System F in this fashion and proved that the encoding is an isomorphism.

Specializing the type of DTC's generic fold operator from Section 2.2

$$\mathit{fold}_D :: \mathit{Functor}\ f \Rightarrow \mathit{Algebra}\ f\ a \to \mathit{Fix}_D\ f \to a$$

for a particular datatype $\mathit{Fix}_D\ f$ yields the type $\mathit{Algebra}\ f\ a \to a$ that we use in Figure 2.11 to define the type-level fixed-point combinator Fix_C for the Church

encoding of that datatype. The generic fold operator $fold_C$ for this fixed-point is simply the application of a value to the given algebra. We can also define one-level folding in_C and unfolding out_C of the fixed-point which are also given in Figure 2.11. These can in turn be used to define new *inject* and *project* functions for the definition of smart constructors and feature specific algebras. DTC's machinery for taking the coproduct of functors and algebras carries over to the new fixed-point unchanged. User-defined algebras for semantic functions only need to be altered to use the new smart constructors.

2.4.2 Reasoning with Church Encodings

The Church encoding of strictly-positive types carries over to (and can be extended in) the Calculus of Constructions (CoC) [Pfenning and Paulin-Mohring, 1990]. However, proper structural induction principles for Church encodings are not provable in CoC [Pfenning and Paulin-Mohring, 1990]. Such induction principles have to be assumed as axioms instead. MTC side-steps this issue and uses a weaker form of induction for which it adapts the proof methods used in the *initial algebra semantics of data types* [Goguen et al., 1977; Malcolm, 1990] – in particular *universal properties* – to support inductive proofs over Church encodings. Consider the type signature of the function *indNat2* that represents an alternative induction principles for the natural numbers:

$$
\begin{array}{l}
\mathit{indNat2} :: \\
\quad \forall P :: \mathit{Nat} \to \mathit{Prop}. \\
\qquad (\mathit{pzero} :: P\ \mathit{Zero}) \to \\
\qquad (\mathit{psucc} :: \forall m :: \mathit{Nat}.P\ m \to P\ (\mathit{Succ}\ m)) \to \\
\qquad \mathit{Algebra}\ \mathit{NatF}\ (\exists m.P\ m)
\end{array}
$$

The induction principle uses a dependent sum type to turn a proof algebra, consisting of the functions *pzero* and *psucc*, into a regular algebra. The algebra builds a copy of the original value and a proof that the property holds for the copy. The proof for the copy can be obtained by folding with this algebra. In order to draw conclusions about the original value two additional *well-formedness* conditions have to be proven.

1. The proof-algebra has to be well-formed in the sense that it really builds a copy of the original value instead of producing an arbitrary value of the same type. This proof needs to be done only once for every induction principle of every functor and is usually short and straightforward.

   In the MTC framework, the well-formedness proof is about 20 LoC per

feature and its use is completely automated using type-classes and hence hidden from the user.

2. The fold operator used to build the proof using the algebra needs to be a proper fold operator, i.e. it needs to satisfy the universal property of folds.

$$\begin{aligned}&\textbf{type } \mathit{UniversalProperty}\ (f :: * \to *)\ (e :: \mathit{Fix}_C\ f)\\ &\quad = \forall a\ (\mathit{alg} :: \mathit{Algebra}\ f\ a)\ (h :: \mathit{Fix}_C\ f \to a).\\ &\qquad (\forall e.h\ (\mathit{in}_C\ e) = \mathit{alg}\ h\ e) \to\\ &\qquad\quad h\ e = \mathit{fold}_C\ \mathit{alg}\ e\end{aligned}$$

In an initial algebra representation of an inductive datatype, we have a single implementation of a fold operator that can be proven correct. In MTC's approach based on Church encodings however, each value consists of a separate fold implementation that must satisfy the universal property.

Hence, in order to enable reasoning MTC must provide a proof of the universal property of folds for every value of a modular datatype that is used in a proof. This is mostly done by packaging a term and the proof of the universal property of its fold in a dependent sum type.

$$\textbf{type } \mathit{FixUP}\ f = \exists(x :: \mathit{Fix}_C\ f).\mathit{UniversalProperty}\ f\ x$$

One of the main novelties of MTC is that this approach to induction also gives us modularity: Proofs are written in the same modular style as functions. These algebras are folded over the terms and can be modularly combined.

2.5 Mendler Folds

MTC encodes data types and folds with a variant of Church encodings based on Mendler folds [Mendler, 1987, 1991; Uustalu and Vene, 2000]. The benefit of this encoding is that we have more control over the evaluation: 1) it allows us to explicitly define the evaluation order of recursive positions instead of relying on the evaluation order of the meta-language and 2) model general-recursive evaluation via bounded fixed-points.

Mendler Church Encodings In comparison to ordinary Church encodings, the Mendler Church encodings ($\mathit{Fix}_M\ f$) differ in their use of Mendler algebras ($\mathit{Algebra}_M\ f\ a$) instead of ordinary F-algebras as shown in Figure 2.12.

$$
\begin{aligned}
&\textbf{type } \mathit{Algebra}_M\ f\ a = \forall r.(r \to a) \to f\ r \to a \\
&\textbf{type } \mathit{Mixin}\ r\ f\ a = (r \to a) \to f\ r \to a \\
&\textbf{type } \mathit{Fix}_M\ f = \forall a.\mathit{Algebra}_M\ f\ a \to a \\
&\mathit{fold}_M :: \mathit{Algebra}_M\ f\ a \to \mathit{Fix}_M\ f \to a \\
&\mathit{fold}_M\ \mathit{alg}\ \mathit{fa} = \mathit{fa}\ \mathit{alg}
\end{aligned}
$$

Figure 2.12: Modular datatypes using Mendler-Church encodings

$$
\begin{aligned}
&\textbf{data } \mathit{Logic}_F\ e = \mathit{BLit}\ \mathit{Bool} \mid \mathit{If}\ e\ e\ e \\
&\textbf{type } \mathit{Value} = \mathit{Bool} \\
&\mathit{ifAlg} :: \mathit{Algebra}_M\ \mathit{Logic}_F\ \mathit{Value} \\
&\mathit{ifAlg}\ [\![\cdot]\!]\ (\mathit{BLit}\ b) \quad = b \\
&\mathit{ifAlg}\ [\![\cdot]\!]\ (\mathit{If}\ e_1\ e_2\ e_3) = \textbf{if } [\![e_1]\!] \textbf{ then } [\![e_2]\!] \textbf{ else } [\![e_3]\!] \\
&\mathit{eval} :: \mathit{Fix}_M\ \mathit{Logic}_F \to \mathit{Value} \\
&\mathit{eval} = \mathit{fold}_M\ \mathit{ifAlg}
\end{aligned}
$$

Figure 2.13: Boolean expressions using Mendler-Church encodings

Mendler algebras take an additional function argument of type $(r \to a)$ for their recursive calls. To enforce structurally recursive calls, arguments which appear at recursive positions have a polymorphic type r. Using this polymorphic type prevents case analysis, or any type of inspection, on those arguments. Mendler folds ($\mathit{fold}_M\ \mathit{fa}\ \mathit{alg}$) are defined by directly applying a Church encoded value *fa* to a Mendler algebra *alg*. All these definitions are non-recursive and can thus be expressed in Coq.

The *Mixin* type slightly generalizes Mendler algebras by using an additional type parameter instead of the universal type quantification. This generalization is useful for defining non-inductive language features such as general recursion or higher-order binders.

Example As a simple example, consider a language for boolean expressions supporting boolean literals and conditionals as shown in Figure 2.13.

The evaluation algebra *ifAlg* for this language takes the function argument $[\![\cdot]\!]$ for evaluating recursive positions. Hence the recursive calls are explicit.

The evaluation function *eval* simply folds the *ifAlg* algebra.

Outlook In the following chapter we present a different approach to encoding fixed-points and folds using universes and datatype-generic programming that overcomes several shortcomings of MTC's use of Church encodings: it is entirely predicative, admits proper strong induction principles and does not require well-formedness proofs for proof algebras.

Chapter 3

Modular Predicative Universes

The *Datatypes à la Carte* (DTC) approach is a Haskell solution for modular programming in Haskell. However, as outlined in Section 2.3 the transition to a proof-assistant, for modular reasoning, comes with major hurdles. DTC relies on a general fixed point combinator to define fixed points for arbitrary functors and uses a generic fold operation that is not structurally recursive. To keep logical consistency Coq applies conservative restrictions and rejects both: a) DTC's type level fixed-points because it cannot see that the definition is always strictly-positive, and b) DTC's fold operator because it cannot determine termination automatically.

Meta-Theory à la Carte (MTC) [Delaware et al., 2013] solves both problems by using extensible Church encodings. However, MTC's use of Church encodings leaves much to be desired. This chapter discusses the problems that Church encodings bring with them in terms of reasoning and presents an alternative implementation of MTC based on a predicative universe of strictly-positive functors instead of Church encodings. The universe admits generic definitions of folds and proper strong induction that fulfill Coq's conservative restrictions.

Outline We first discuss the drawbacks of using Church encodings in Section 3.1 to motivate their replacement. Section 3.2 presents the approach to modularized induction for our new representation. This results in a different user-facing interface than MTC's for modular reasoning. In particular, the representation of *proof algebras* is significantly different from the one used with

MTC's weak induction principle and closer to standard structural inductive reasoning. We discuss the universe of container types in Section 3.5 which is modular and which admits datatype-generic generic definitions of folds and induction that we use to generically instantiate our interface. The universe of containers is very large and admits only a small number of generic functions. As a complement, we discuss the universe of polynomial functors in Section 3.6, which admits more generic functions like generic equality, and show how to embed it in the universe of containers which provides us with additional reuse opportunities. We compare the universe basec solution directly with MTC's Church encoding solution in Section 3.7 using a port of MTC's case study.

3.1 Motivation

The MTC framework uses Church encodings to represent modular datatypes. Unfortunately Church encodings have multiple drawbacks when it comes to reasoning:

1. Church encodings are inherently impredicative and thus MTC has to rely on on an impredicative sort. Hence it is forced to use Coq's `impredicative-set` option. However, this option is inconsistent with standard axioms of classical logic like the law of excluded middle and double negation elimination. This also restricts the approach to systems that allow impredicative encodings and hence rules out systems that are fully predicative like Agda.

2. The fixed-point combinator provided by Church encodings admits too many functors. For inductive reasoning, only strictly-positive functors are valid, i.e, those functors whose fixed-points are inductive datatypes. Yet, Church encodings do not rule out other functors. Hence, in order to reason only about inductive values, MTC requires a witness of inductivity: the universal property of folds. Since every value comes with its own implementation of the fold operator, MTC needs to keep track of a different such witness for every value. It does so by decorating each value with its witness with the help of a Σ-type.

   As a result of this decoration, the user is confronted with a mix of decorated and un-decorated values. This obviously impairs the readability of the code, but also creates confusion about which variant is the proper one to use when stating propositions. Moreover, since proofs are opaque in Coq, it also causes problems for equality of terms. Finally, the decoration makes it unclear whether MTC adequately encodes fixed-points.

3. Böhm and Berarducci's isomorphism of strictly-positive datatypes and their Church encodings in System F [Böhm and Berarducci, 1985] is a purely meta-theoretic result. Hence, we have the guarantee that an induction principle exists for Church encodings of strictly-positive datatypes, yet Coq's theory is not powerful enough to prove this for Church encodings directly expressed in Coq.

   MTC relies on a *poor-man's induction principle* instead and requires the user to provide additional well-formedness proofs. Even though these can be automated with proof tactics, they nevertheless complicate the use of the framework.

We take an alternative approach by applying well-known datatype-generic programming (DGP) techniques to represent modular datatypes, to build functions from functor algebras with generic folds and to compose proofs from proof algebras by means of generic induction. This overcomes the above shortcomings:

1. It does not assume `impredicative-set` or any axioms other than standard functional extensionality.

2. A witness of inductivity is always associated with the type, i.e. a typeclass instance that holds the universe code for a functor, and not with values.

3. The generic induction principle is a proper one that does not rely on any additional well-formedness conditions. Moreover, for some functionality and proofs, our approach can achieve more reuse than MTC: instead of composing modular components we provide a single generic definition once and for all.

Another difference with MTC is that we split the solution into a frontend and a backend part . Both parts are connected via a declarative specification of *functors*, *fixed-points*, *folds* and *induction principles*. The frontend extends the specification with support for modularity to form the user-facing interface and the backend is an implementation of the specification in terms of container types.

3.2 Declarative Specification

Similar to DTC and MTC our approach relies on fixed-points of functors to model datatypes, folds to implement functions on datatypes and on abstraction

```
class Functor f ⇒ PFunctor f where
  type All          :: ∀a.(a → Prop) → f a → Prop
  all_fmap          :: ∀a b (g :: a → b) (p :: b → Prop) (xs :: f a).
                         All f a p (fmap g xs) ↔
                         All f a (λx → p (g x)) xs
type PAlgebra f a (alg :: Algebra f a) (p :: a → Prop) =
  PFunctor f ⇒ ∀(xs :: f a).All f a p xs → p (alg xs)
class PFunctor f ⇒ SPF (f :: ∗ → ∗) where
    -- Fixed-points
  type Fix_F        :: ∗
  in_F              :: f (Fix_F f) → Fix_F f
  out_F             :: Fix_F f → f (Fix_F f)
  in_out_inverse    :: ∀(e :: Fix_F f).      in_F (out_F e) = e
  out_in_inverse    :: ∀(e :: f (Fix_F F)).out_F (in_F e) = e
    -- Folds
  fold              :: ∀a.Algebra f a → Fix_F f → a
  fold_uniqueness   :: ∀a (alg :: Algebra f a) h.
                         (∀e.h (in_F e) = alg (fmap h e)) →
                         ∀x.h x = fold alg x
  fold_computation :: ∀a (alg :: Algebra f a) (x :: a),
                         fold alg (in_F x) = alg (fmap (fold alg) x)
    -- Induction
  ind               :: ∀(p :: Fix_F f → Prop).PAlgebra in_F p → ∀e.p e
```

Figure 3.1: Strictly-positive functor class

over *super-functors* and *super-algebras* to achieve modularity in programming and reasoning. We do not lump all of these concepts together in one interface because the modular composition of signature functor, function algebras and proof algebras is not an essential part of the fixed-point construction. The only concern for the fixed-point construction in our interface is the support for modularity through opening up the recursion. We therefore separate the code concerning fixed-points into a backend layer and abstract over its implementation by defining a declarative specification of fixed-points and related definitions of algebras, folds, proof algebras and induction. This section describes the *declarative specification* and Section 3.5 presents a backend implementa-

tion based on container types. The user-facing frontend differs from MTC mainly in the use of *modular proof algebras* and *modular induction principles*. These differences are discussed in Section 3.4.

The *SPF* type class in Figure 3.1 is a core part of the interface that serves as a declarative specification of our requirements on functors and carries the required evidence. We discuss each concept that appears in the type class in turn starting with the programming related parts.

3.2.1 Fixed-Points

While we need the existence of a fixed-point type of *abstract super-functors*, it is inessential how this is constructed. This means that instead of providing a generic fixed-point type constructor like Fix_D we can alternatively provide a witness of the existence of a valid fixed-point in the type class, i.e. we make the fixed-point an associated type of the *SPF* type class. We thereby delay the problem of defining the fixpoint until the final signature functor composition is created. At this point the user can either use the generic fixed-point combinator that we define in Section 3.5 or even define his own. *SPF* also includes the initial algebra function in_F and its inverse out_F as members to fold/unfold one layer of the fixed-point. Furthermore, the members *in_out_inverse* and *out_in_inverse* are witnesses that folding/unfolding of the fixpoint type form inverse operations.

3.2.2 Fold Operator

SPF is a subclass of *Functor* so we would like to define a generic fold operator similar to DTC's operator $fold_D$ from Section 2.2.

$$fold_F :: SPF\ f \Rightarrow Algebra\ f\ a \rightarrow Fix_F\ f \rightarrow a$$
$$fold_F\ alg = alg \circ fmap\ (fold_F\ alg) \circ out_F$$

Unfortunately, this definition is not structurally recursive and Coq is not able to determine its termination automatically. Hence, this definition is rejected. This is similar to the problem of Fix_F. For any concrete functor we can inline the definition of *fmap* to let $fold_F$ pass the termination check, but again we are working with an abstract functor f and an abstract functorial mapping *fmap*. We resolve this similarly by including a witness for the existence of a valid fold operator in the *SPF* class and also witnesses that the fold operator satisfies the universal property of folds.

3.3 Declarative Specification of Induction

The *SPF* typeclass also provides an interface for inductive reasoning in terms of an induction principle. In general, the type of an induction principle depends on the number of constructors of a datatypes and their arities which makes a generic definition difficult.

For example, consider the induction principle ind_A for arithmetic expressions:

$$\begin{aligned} ind_A :: \forall(p &:: Arith \to Prop). \\ &\forall(hl :: \forall n. \quad p\ (Lit\ n))). \\ &\forall(ha :: \forall x\ y.p\ x \to p\ y \to p\ (Add\ x\ y))). \\ &\forall(x :: Arith).p\ x \end{aligned}$$

It takes a proposition p as parameter and inductive steps hl and ha for each case. We say that hl and ha together form a *proof algebra* of p. An inductive step consists of showing that p is preserved during one level of construction of a value, i.e. showing that p holds for an application of a constructor given proofs of p for all recursive positions. In case of a literal we have no recursive positions and in case of addition we have two. Proof algebras for other datatypes differ in the number of cases and the number of recursive positions.

For a generic definition of induction, we first need to develop a *uniform representation of induction* which effectively boils down to developing a *uniform representation of proof algebras* which is the subject of the remainder of this section.

3.3.1 All-Modalities

We first focus on the inputs of the proof algebra functions, i.e. the proofs that the induction predicate holds for recursive positions. We use an *all-modality* [Benke et al., 2003; Morris, 2007] for signature functors to capture these proofs. Informally, the all-modality of a functor f and a predicate $(p :: a \to Prop)$ is a new type $(All\ a\ p{::}f\ a \to Prop)$ that denotes that the predicate p holds for each $(x :: a)$ in an $(f\ a)$.

Example: Arithmetic Expressions The following type $Arith_{All}$ is an example of an all-modality for the signature functor $Arith_F$ of arithmetic expressions. The constructor $ALit$ encodes that the all-modality holds for literals and $AAdd$ encodes that the all-modality holds for $(Add\ x\ y)$ if p holds for both recursive positions x and y.

$$\begin{aligned}&\textbf{data}\ Arith_{All}\ a\ p :: Arith_F\ a \to Prop\ \textbf{where}\\ &\quad ALit\ \ ::\qquad\qquad Arith_{All}\ a\ p\ (Lit_F\ n)\\ &\quad AAdd :: p\ x \to p\ y \to Arith_{All}\ a\ p\ (Add_F\ x\ y)\end{aligned}$$

Using the all-modality definition we can write ind_A equivalently as

$$\begin{aligned}&ind_A\prime :: \forall(p :: Arith \to Prop).\\ &\qquad \forall(h :: \forall(xs :: Arith_F\ Arith).Arith_{All}\ a\ p\ xs \to p\ (in_{Arith}\ xs)).\\ &\qquad \forall(x :: Arith).p\ x\end{aligned}$$

The induction principle now takes a *single argument* h that represents the *proof algebra* independent of the number of cases and arity of constructors. Notice in particular the result of h. The constructor applications in the result of the proof algebra functions of ind_A are now combined into a single application of the initial algebra in_{Arith} of $Arith_F$ with carrier $Arith$:

$$\begin{aligned}&in_{Arith} :: Arith_F\ Arith \to Arith\\ &in_{Arith}\ (Lit_F\ n) \quad = Lit\ n\\ &in_{Arith}\ (Add_F\ x\ y) = Add\ x\ y\end{aligned}$$

Comparison to MTC The all-modality $Arith_{All}$ shares the structure of its functor $Arith_F$, reminiscent of ornamentation [McBride, 2010]. In fact, we can represent it using the functor $Arith_F$ as witnessed by the following isomorphism:

$$\begin{aligned}&\forall(a :: *)\ (p :: a \to Prop).\\ &\quad (\exists\ (xs :: Arith_F\ a).Arith_{All}\ a\ p\ xs) \cong (Arith_F\ (\exists\ (x :: a).p\ x))\end{aligned}$$

If access to the index xs is needed, as for example for the induction principles, we can relate the existentially quantified values via an equation:

$$\begin{aligned}&\forall(a :: *)\ (p :: a \to Prop)\ (xs :: Arith_F\ a).\\ &\quad (Arith_{All}\ a\ p\ xs) \cong (\exists\ (ps :: Arith_F\ (\exists\ x.p\ x)).fmap\ projT_1\ ps \equiv xs)\end{aligned}$$

where $(projT_1 :: (\exists\ (x :: a).p\ x) \to a)$ projects a Σ-type to its first component. This suggests, that we can define all-modalities generically without requiring the definition of a separate type. Indeed, MTC uses the right-hand sides of both of the above isomorphisms:

1. The first, existentially quantified variant is used generally for proof algebras. This is a choice that follows directly from MTC's weak induction principle. The constraint on the existential values is proved

by means of a intricate well-formedness requirement for proof algebras ($palg :: Algebra\ Arith_F\ (\exists\ x.p\ x)$):

$$\forall(xs :: Arith_F\ (\exists\ x.p\ x)).\\ projT_1\ (palg\ xs) \equiv inject\ (fmap\ projT_1\ xs)$$

which expresses that the algebra behaves like a sub-algebra of the

initial algebra. This well-formedness proof can be done once for a signature functor and subsequently reused for any proof algebra of that functor, but, the user is still required to keep track of well-formedness properties.

2. The second, equationally constrained variant is used to track the universal property of recursive positions. Unfortunately, MTC does not use *all-modalities* as an *abstract concept* and simply works with the generic definition directly. As a consequence, often both the decorated value *ps* and the undecorated one *xs* are in scope, creating additional noise for the user.

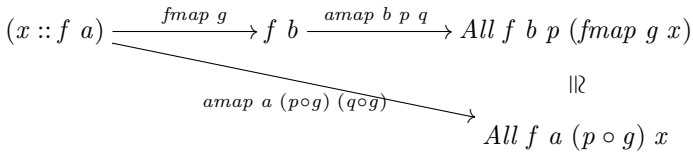
PFunctor Class To counter the proliferation of Σ-types and projections out of Σ-types we do not introduce a generic definition of an *all-modality* in our interface and work with an abstraction instead. To this end, we introduce a new typeclass *PFunctor* that carries the associated all-modality type and make *SPF* a subclass of it.

All-modalities share the structure of their associated functors. For example, the mapping of a functor f generalizes to a dependent variant:

$$amap :: \forall(a :: *)\ (p :: a \to Prop).(\forall(x :: a).p\ x) \to (\forall xs.All\ f\ a\ p\ xs)$$

The function *amap* can be used to define an induction operator in the same way that *fmap* can be used to define a fold operator. However, the same caveats apply: it is not obvious that this is a terminating definition. We adopt a similar solution as for *fold*: inline *amap* in the definition of the induction operator. Hence, because we have no use for *amap* other than in the induction, it is unnecessary to include it in the interface.

We include however one property *all_fmap* that is underlying the generalization of the fmap fusion law:



all_fmap expresses the propositional equivalence of the types on the right side, albeit without a proof that these form an isomorphism. This property is used to derive another induction principle on pairs instead of single values which in turn is used to encode proof algebras of properties of equality functions.

3.3.2 Proof Algebras

In the *Arith* example, the induction principle *ind* A' now takes a uniformly represented proof algebra as a single parameter h. Note that h shows that p holds for an application of the initial algebra in_{Arith}. In the modular setting however, we want to provide proofs for sub-algebras of the initial algebra, or more generally, of any (not necessarily initial) algebra.

As an example, consider the combined arithmetic and logical expressions from Figure 2.5 in Section 2.2 with signature functor $(Arith_F \oplus Logic_F)$. The induction principle for the non-modular datatype *Exp* has the type

$$
\begin{array}{l}
indExp :: \forall(p :: Exp \to Prop). \\
\qquad \forall(hl :: \forall n. \qquad\qquad\qquad\quad p\ (Lit\ n))). \\
\qquad \forall(ha :: \forall x\ y.\ \ p\ x \to p\ y \to \quad p\ (Add\ x\ y))). \\
\qquad \forall(hb :: \forall b. \qquad\qquad\qquad\quad p\ (BLit\ b))). \\
\qquad \forall(hi :: \forall x\ y\ z. p\ x \to p\ y \to p\ z \to p\ (If\ x\ y\ z))). \\
\qquad \forall(x :: Exp). p\ x
\end{array}
$$

For the purpose of modularity, we want to represent the proof algebras of specific features, i.e. signature sub-functors, separately and combine these *proof sub-algebras* to a complete proof algebra for the initial algebra. The result type of proof sub-algebras needs to be a value of the fixed-point type. Hence, we inject the signature sub-functor, e.g. $Arith_F$, into the complete signature functor, e.g. $(Arith_F \oplus Logic_F)$, and then apply the initial algebra; this is exactly what is performed by *inject*. We can thus rewrite the above induction principle into one which uses the uniform representation for each feature.

```
instance (PFunctor f, PFunctor g) ⇒ PFunctor (f ⊕ g) where
    type All a p xs = case xs of
                        Inl xs → All a p xs
                        Inr xs → All a p xs
    all_fmap = ...
class PFunctor f ⇒
    ProofAlgebra f a (alg :: Algebra f a) (p :: a → Prop)
  where
    palgebra :: PAlgebra f a alg p
instance (ProofAlgebra f a falg p, ProofAlgebra g a galg p) ⇒
  ProofAlgebra (f ⊕ g) a (algebraPlus falg galg) p where
    palgebra (Inl xs)  axs = palgebra xs axs
    palgebra (Inr xs)  axs = palgebra xs axs
```

Figure 3.2: Modular composition of proofs

$$
\begin{aligned}
indExp' :: \forall(p :: Exp \to Prop).\ & \\
& \forall(ha :: \forall(xs :: Arith_F\ Exp).Arith_{All}\ Exp\ p\ xs \to p\ (inject\ xs)).\\
& \forall(hl\ :: \forall(xs :: Logic_F\ Exp).Logic_{All}\ Exp\ p\ xs \to p\ (inject\ xs)).\\
& \forall(x :: Exp).p\ x
\end{aligned}
$$

We will discuss how the proof sub-algebras can be composed into a proof-algebra for the intial algebra in Section 3.4.

3.3.3 Induction Operator

As discussed above, just like with *fold*, the generic definition of the induction operator for abstract functors is not structurally recursive and we apply a similar solution to solve it: we delay the problem of defining induction to the point where the final composition is made and require its existence by adding an induction operator *ind* as a member of the *SPF* class. The *ind* operator takes a property p of a and a proof algebra for the initial algebra and constructs a proof for every value of Fix_F.

3.4 Modularity Frontend

The modular composition of signatures and semantic functions in our approach, based on co-products of functors, is the same as in DTC and MTC and carries over largely unchanged to our declarative specification. Therefore we discuss only the composition of modular proofs in this section.

Figure 3.2 contains the instance of the *PFunctor* class for a co-product $(f \oplus g)$. For both, the associated type *All* and the property *all_fmap* a simple case distinction is sufficient.

We use the type class *ProofAlgebra*, also shown in Figure 3.2, to define and assemble proof algebras in a modular fashion. The parameter f represents the underlying functor, a the carrier type, *alg* the underlying f-algebra and p a property of the carrier.

In the definition of the *ProofAlgebra* instance for functor composition we use the function *algebraPlus* from Figure 2.9 in Section 2.2.3 to composes the two function algebras *falg* and *galg* which also forms the implementation of the *FAlgebra* instance for co-products. To avoid any coherence concerns, we assume that algebras are always composed using *algebraPlus* – or equivalently the *FAlgebra* instance for composition.

3.4.1 Non-Modularity of SPF

When instantiating modular functions to a specific set of signatures, we need an *SPF* instance for the coproduct of that set. Ideally, as with algebras, we would like to derive an instance for $f \oplus g$ given instances for f and g, because we cannot expect the programmer to provide an instance for every possible set of signatures.

Unfortunately, *SPF* does not include enough information about the functors to do this in a constructive way. We cannot construct the fixed-point of the coproduct $f \oplus g$ from the fixed-points of the summands f and g and likewise for the fold and induction operators. Therefore *SPF* serves solely as a high-level interface class.

In Section 3.5 we develop an approach that side-steps this issue. Instead of composing fixed-points, folds and induction along coproducts of arbitrary *SPF*s, we focus on the class of containers which are strictly-positive functors that are 1) closed under coproducts and 2) allow a generic instantiation of *SPF*'s interface.

```
data Arith_F a = Lit_F Nat | Add_F a a
  deriving Functor
data Arith_All (a :: *) (p :: a → *) :: Arith_F a → *where
  ALitF :: ∀(n :: Nat).Arith_All a p (Lit_F n)
  AAddF :: ∀(a_1 a_2 :: a).p a_1 → p a_2 → Arith_All a p (Add_F a_1 a_2)
instance PFunctor Arith_F where
  type All Arith_F a = Arith_All a
  all_fmap = ...
```

```
data Logic_F a = BLit_F Bool | If_F a a a
  deriving Functor
data Logic_All (a :: *) (p :: a → *) :: Logic_F a → *where
  ABLitF :: ∀(b :: Bool).Arith_All a p (BLit_F b)
  AIfF :: ∀(i t e :: a).p i → p t → p e → Arith_All a p (If_F i t e)
instance PFunctor Logic_F where
  type All Logic_F a = Logic_All a
  all_fmap = ...
```

Figure 3.3: Arithmetic and logical expressions

3.4.2 Example: Depth vs. Size

In this section we develop a complete example to showcase how the previous definitions work. We reuse one of [Pierce, 2002] basic examples of structural induction: Define a *depth* and a *size* function on expressions and show that the depth is always smaller than the size. We compose expressions out of two features that we define independently: arithmetic and logical expressions. Figure 3.3 shows the signature functors $Arith_F$ (top) and $Logic_F$ (bottom) for the two features, their all-modalities and *PFunctor* instances.

To avoid giving away the generic definition of fixed-points, folds and induction from Section 3.5, we simply instantiate the *SPF* class manually with the non-modular datatype definition of expressions that contains both arithmentic and logical expressions. The datatype and the *SPF* instance are show in Figure 3.4.

Figure 3.5 shows the modular definition of the two semantic functions *depthOf* and *sizeOf*. Each feature/function combination has a separate named

```
type Exp_F = Arith_F ⊕ Logic_F
data Exp = Lit Nat | Add Exp Exp | BLit Bool | If Exp Exp Exp
instance SPF Exp_F where
  type Fix_F Exp_F = Exp
  in_F (Inl (Lit_F n))       = Lit n
  in_F (Inl (Add_F a1 a2)) = Add a1 a2
  in_F (Inr (BLit_F b))      = BLit b
  in_F (Inr (If_F i t e))    = If i t e
  out_F (Lit n)       = Inl (Lit_F n)
  out_F (Add a1 a2) = Inl (Add_F a1 a2)
  out_F (BLit b)      = Inr (BLit_F b)
  out_F (If i t e)    = Inr (If_F i t e)
  inoutinverse = ...
  outininverse = ...
  fold alg (Lit n)       = alg (Inl (Lit_F n))
  fold alg (Add a1 a2) = alg (Inl (Add_F (fold alg a1) (fold alg a2)))
  fold alg (BLit b)      = alg (Inr (BLit_F b))
  fold alg (If i t e)    =
    alg (Inr (If (fold alg i) (fold alg t) (fold alg e)))
  folduniqueness = ...
  foldcomputation = ...
  ind = ...
```

Figure 3.4: *SPF* instance for expressions

FAlgebra instance and the semantic functions themselves are defined as a fold over the fixed-point of any signature functor for which an *FAlgebra* exists, including the combined Exp_F signature functor of arithmetic and logical expressions.

Similarly, Figure 3.6 defines proof algebra instances of *DepthSize* for each feature separately. We omit the proofs for brevity. The final proof *depthSize* is again overloaded: we get the property for the fixed-point of any signature functor with a *DepthSize* proof algebra instance.

```
data DepthOf = DepthOf
depthOf :: (SPF f, FAlgebra DepthOf f Nat) ⇒ Fix_F f → Nat
depthOf = fold (falgebra DepthOf)
instance FAlgebra DepthOf Arith_F Nat where
  falgebra DepthOf (Lit_F n)       = 0
  falgebra DepthOf (Add_F a_1 a_2) = 1 + max a_1 a_2
instance FAlgebra DepthOf Logic_F Nat where
  falgebra DepthOf (BLit_F n)      = 0
  falgebra DepthOf (If_F i t e)    = 1 + max i (max t e)
```

```
data SizeOf = SizeOf
sizeOf :: (SPF f, FAlgebra SizeOf f Nat) ⇒ Fix_F f → Nat
sizeOf = fold (falgebra SizeOf)
instance FAlgebra SizeOf Arith_F Nat where
  falgebra SizeOf (Lit_F n)       = 1
  falgebra SizeOf (Add_F a_1 a_2) = 1 + a_1 + a_2
instance FAlgebra SizeOf Logic_F Nat where
  falgebra SizeOf (BLit_F n)      = 0
  falgebra SizeOf (If_F i t e)    = 1 + i + t + e
```

Figure 3.5: Modular semantic functions

3.5 Containers

The type-class *SPF* of Section 3.2 captures all the requirements on abstract functors for modular programming. We can modularly compose algebras and proof algebras for semantics functions and proofs. However, as discussed in Section 3.4.1 *SPF* itself is not modular in the sense that we cannot construct coproducts (directly). In the example in Section 3.4.2 we avoided that issue by manually giving the instance of the *SPF* class for the sum of the signature functors $Arith_F$ and $Logic_F$. This is essentially the approach taken by [Schwaab and Siek, 2013].

In this section we go the last mile and implement a modular refinement of *SPF* using datatype-generic programming (DGP) in general and containers in particular. The problem of defining fixed-points for a class of functors also arises in many approaches to DGP and we can use the same techniques in our

```
type DepthSize e = depthOf e < sizeOf e
depthSize :: (SPF f, FAlgebra DepthOf f Nat,
  FAlgebra SizeOf f Nat ProofAlgebra f (Fix_F f) in_F DepthSize) ⇒
  ∀(e :: Fix_F f).DepthSize e
depthSize = ind f DepthSize (palgebra f (Fix_F f) in_F DepthSize)
instance (SPF f, ...) ⇒
    ProofAlgebra Arith_F (Fix_F f) inject DepthSize
  where
    palgebra (ALitF n)           = ...
    palgebra (AAddF a1 a2 p1 p2) = ...
instance (SPF f, ...) ⇒
    ProofAlgebra Logic_F (Fix_F f) inject DepthSize
  where
    palgebra (ABLitF b)          = ...
    palgebra (AIfF a1 a2 p1 p2)  = ...
```

Figure 3.6: Modular *DepthSize* proof

setting. Containers are one approach to DGP that models a class of functors which is 1) closed under coproducts 2) and admits a generic implementation of *SPF*'s methods that respects all the restrictions of the proof-assistant setting.

Section 3.5.1 discusses universes in general and Section 3.5.2 and Section 3.5.2 reviews the universe of containers in particular. Sections 3.5.3, 3.5.4 and 3.5.5 discuss the implementation of coproducts, fixed-points & folds and induction respectively. Finally in Section 3.5.6 we bridge the gap to modular programming. We show how *Functor*, *PFunctor* and *SPF* are instantiated by containers and discuss the automation for composing the container instances of a set of signature functors.

3.5.1 Generic Universes

In a dependently-typed setting it is common to use a universe for generic programming [Altenkirch and McBride, 2003; Altenkirch et al., 2007; Benke et al., 2003]. A universe consists of two important parts:

1. A set *Code* of codes that represent types in the universe.
2. An interpretation function *Ext* that maps codes to types.

```
data Cont where
  (▷) :: (s :: ∗) → (p :: s → ∗) → Cont
shape (s ▷ p) = s
pos   (s ▷ p) = p
data Ext (c :: Cont) (a :: ∗) where
  Ext :: (s :: shape c) → (pos c s → a) → Ext c a
```

Figure 3.7: Container extension

There is a large number of approaches to DGP that vary in the class of types they can represent and the generic functions they admit. For our application we choose the universe of containers [Abbott et al., 2005].

An important property of the container universe is that it can represent all strictly-positive functors [Abbott et al., 2005] and allows folds and induction to be implemented generically. Hence, we meet our goal and do not loose any expressivity.

In Section 3.6 we discuss the universe of polynomial functors. it is a subuniverse of containers in the sense that any polynomial functor is also a container, but the universe admits more generic functions. We use this universe to supplement our approach with a generic implementation of equality in Section 3.6.3 to achieve more reuse.

3.5.2 Container Universe

The codes of the container universe are of the form $S \triangleright P$ where S denotes a type of shapes and $P :: S \to *$ denotes a family of position types indexed by S. The extension $Ext\ c$ of a container c in Figure 3.7 is a functor. A value of the extensions $Ext\ c\ a$ consists of a shape $s :: shape\ c$ and for each position $p :: pos\ c\ s$ of the given shape we have a value of type a. We can define the functorial mapping *gfmap* generically for any container.

```
gfmap :: (a → b) → Ext c a → Ext c b
gfmap f (Ext s pf) = Ext s (λp → f (pf p))
```

Example The functor $Arith_F$ for arithmetic expressions can be represented as a container functor using the following shape and position type.

$$
\begin{aligned}
&\textbf{data}\ Arith_S = Lit_S\ Int \mid Add_S\\
&\textbf{data}\ Arith_P :: Arith_S \to *\textbf{where}\\
&\quad Add_{P1} :: Arith_P\ Add_S\\
&\quad Add_{P2} :: Arith_P\ Add_S\\
&\textbf{type}\ Arith_C = Arith_S \triangleright Arith_P
\end{aligned}
$$

The shape of an $Arith_F$ value is either a literal Lit with some integer value or it is an addition Add. In case of Add we have two recursive positions Add_{P1} and Add_{P2}. Lit does not have any recursive positions.

The isomorphism between $Arith_F$ and $Ext\ Arith_C$ is witnessed by the following two conversion functions.

$$
\begin{aligned}
&from :: Arith_F\ a \to Ext\ Arith_C\ a\\
&from\ (Lit\ i) = Ext\ (Lit_S\ i)\ (\lambda p \to \textbf{case}\ p\ \textbf{of}\ \{\})\\
&from\ (Add\ x\ y) = Ext\ Add_S\ pf\\
&\quad \textbf{where}\ pf :: Arith_P\ Add_S \to a\\
&\qquad\quad pf\ Add_{P1} = x\\
&\qquad\quad pf\ Add_{P2} = y\\
&to :: Ext\ Arith_C\ a \to Arith_F\ a\\
&to\ (Ext\ (Lit_S\ i)\ pf) = Lit\ i\\
&to\ (Ext\ Add_S\quad pf) = Add\ (pf\ Add_{P1})\ (pf\ Add_{P2})
\end{aligned}
$$

Literals do not have recursive positions and hence we cannot come up with a position value. In Coq one needs to refute the position value $p :: Arith_P\ (Lit\ i)$ as its type is uninhabited. We use a case distinction without alternatives as an elimination.

3.5.3 Coproducts

Given two containers $S_1 \triangleright P_1$ and $S_2 \triangleright P_2$ we can construct a coproduct. The shape of the coproduct is given by the coproducts of the shape and the family of position types delegates the shape to the families P_1 and P_2. Figure 3.8 contains the definitions of shape and positions of the coproduct and injection functions on the extensions.

3.5.4 Fixpoints and Folds

The universe of containers allows multiple generic constructions. First of all, the fixpoint of a container is given by its W-type.

$$
\textbf{data}\ W\ (c :: Cont) = Sup\ \{\, unSup :: Ext\ c\ (W\ c)\,\}
$$

$$
\begin{array}{l}
\textbf{type}\ S_+ = \mathit{Either}\ S_1\ S_2 \\
\textbf{type}\ P_+\ (\mathit{Left}\ s) \ \ = P_1\ s \\
\textbf{type}\ P_+\ (\mathit{Right}\ s) = P_2\ s \\
\mathit{inl} :: \mathit{Ext}\ (S_1 \triangleright P_1) \to \mathit{Ext}\ (S_+ \triangleright P_+) \\
\mathit{inl}\ (\mathit{Ext}\ s\ pf) = \mathit{Ext}\ (\mathit{Left}\ s)\ pf \\
\mathit{inr} :: \mathit{Ext}\ (S_2 \triangleright P_2) \to \mathit{Ext}\ (S_+ \triangleright P_+) \\
\mathit{inr}\ (\mathit{Ext}\ s\ pf) = \mathit{Ext}\ (\mathit{Right}\ s)\ pf
\end{array}
$$

Figure 3.8: Container coproducts

The definition of *Ext* is known at this point and Coq can see that the $W\ c$ is strictly positive for any container c and hence the definition of W is accepted.

Furthermore, we define a fold operator generically.

$$
\begin{array}{l}
\mathit{gfold} :: \mathit{Algebra}\ (\mathit{Ext}\ c)\ a \to W\ c \to a \\
\mathit{gfold}\ \mathit{alg}\ (\mathit{Sup}\ (\mathit{Ext}\ s\ pf)) = \\
\quad \mathit{alg}\ (\mathit{Ext}\ s\ (\lambda p \to \mathit{gfold}\ \mathit{alg}\ (pf\ p)))
\end{array}
$$

We have obtained this definition by taking the usual definition

$$
\begin{array}{l}
\mathit{gfold}\ \mathit{alg}\ x = \\
\quad \mathit{alg}\ (\mathit{gfmap}\ (\mathit{gfold}\ \mathit{alg})\ (\mathit{unSup}\ x))
\end{array}
$$

which is essentially the same as the definition of fold_D from Section 2.2 and inlining the implementation of *gfmap*. Because this exposes the structural recursion, Coq accepts the definition. Indeed the recursive call $\mathit{gfold}\ \mathit{alg}\ (pf\ p)$ is performed on the structurally smaller argument $pf\ p$. Note that, unlike for fold_D, inlining is possible because *gfmap* is defined uniformly for all containers.

3.5.5 Induction

To define an induction principle for container types we proceed in the same way as in Section 3.3 by defining proof algebras using an *all-modality* [Benke et al., 2003]. The all-modality on containers is given generically by a Π-type that asserts that q holds at all positions as shown in Figure 3.9.

As with the implementation of the generic fold operations, enough structure is exposed to write a valid induction function: *gind* calls itself recursively on the structurally smaller values $pf\ p$ to establish the proofs of the recursive positions before applying the proof algebra *palg*.

$$
\begin{aligned}
&GAll :: (q :: a \to Prop) \to Ext\ c\ a \to Prop \\
&GAll\ q\ (Ext\ s\ pf) = \forall (p :: pos\ c\ s).q\ (pf\ p) \\
&gind :: \forall (c \quad :: Cont) \to \\
&\qquad \forall (q \quad :: W\ c \to Prop) \to \\
&\qquad \forall (palg :: \forall xs.GAll\ q\ xs \to q\ (Sup\ xs)) \to \\
&\qquad \forall x.q\ x \\
&gind\ c\ q\ palg\ (Sup\ (Ext\ s\ pf)) = \\
&\quad palg\ (\lambda p \to gind\ c\ q\ palg\ (pf\ p))
\end{aligned}
$$

Figure 3.9: Container induction

$$
\begin{aligned}
&\textbf{class}\ Container\ (f :: * \to *)\ \textbf{where} \\
&\quad cont \quad :: Cont \\
&\quad from \quad :: f\ a \to Ext\ c\ a \\
&\quad to \quad :: Ext\ c\ a \to f\ a \\
&\quad fromTo :: \forall x.from\ (to\ x) \equiv x \\
&\quad toFrom :: \forall x.to\ (from\ x) \equiv x
\end{aligned}
$$

Figure 3.10: Container functor class

3.5.6 Container Class

Directly working with the container representation is cumbersome for the user. As a syntactic convenience we allow the user to use any conventional functor of type $* \to *$ as long as it is isomorphic to a container functor. The type class *Container* in Figure 3.10 witnesses this isomorphism. The class contains the functions *from* and *to* that perform the conversion between a conventional functor and a container functor and proofs that these conversions are inverses.

Via the isomorphisms *from* and *to* we can import all the generic functions to concrete functors and give instances for *Functor*, *PFunctor*, and *SPF*, which are displayed in Figure 3.11.

The important difference to the *SPF* class is that we can generically build the instance for the coproduct of two *Container* functors

$$\textbf{instance}\ (Container\ f, Container\ g) \Rightarrow Container\ (f \oplus g)$$

```
instance Container f ⇒ Functor f where
  fmap f    = to ∘ gfmap f ∘ from
instance Container f ⇒ PFunctor f where
  All q     = GAll q ∘ from
  all_fmap = ...
instance Container f ⇒ SPF f where
  Fix_F     = W S P
  in_F      = sup ∘ from
  out_F     = to ∘ unSup
  fold alg  = gfold (alg ∘ to)
  ...
```

Figure 3.11: Container instances

by using the coproduct of their containers with the generic coproduct construction from Section 3.5.3.

3.5.7 Extensible Inductive Relations

Many properties are expressed as relations over datatypes. These relations are represented by inductive families where a constructor of the family corresponds to a rule defining the relation.

When using relations over extensible datatypes the set of rules must be extensible as well. For instance, a well-typing relation of values $WTValue :: (Value, Type) \to Prop$ must be extended with new rules when new cases are added to $Value$.

Extensibility of inductive families is obtained in the same way as for inductive datatypes by modularly building inductive families as fixpoints of functors between inductive families. The following indexed functor $WTNat_F$ covers the rule that a natural number value has a natural number type.

```
data WTNat_F (wfv :: (Fix_F vf, Fix_F tf) → Prop) ::
  (Value, Type) → Prop where
  WTNat :: (NatValueF ≺: vf, NatTypeF ≺: tf) ⇒
    WTNat wfv (vi n, tnat)
```

MTC constructs fixed points of indexed functors also by means of Church encodings. The indexed variants of algebras and fixed points are

$$
\begin{array}{l}
\textbf{class}\ \mathit{IFunctor}\ i\ (f :: (i \to \mathit{Prop}) \to i \to \mathit{Prop})\ \textbf{where} \\
\quad \mathit{ifmap} :: \forall (a :: i \to \mathit{Prop})\ (b :: i \to \mathit{Prop})\ (j :: i). \\
\qquad\qquad (\forall j.a\ j \to b\ j) \to f\ a\ j \to f\ b\ j \\
\textbf{class}\ \mathit{IFunctor}\ i\ f \Rightarrow \\
\qquad \mathit{ISPF}\ i\ (f :: (i \to \mathit{Prop}) \to i \to \mathit{Prop})\ \textbf{where} \\
\quad \textbf{type}\ \mathit{IFix} :: i \to \mathit{Prop} \\
\quad \mathit{in}_{IF} \quad :: \forall (j :: i).f\ (\mathit{IFix}\ f\ i) \to \mathit{IFix}\ f\ i \\
\quad \mathit{out}_{IF} \quad :: \forall (j :: i).\mathit{IFix}\ f\ i \to f\ (\mathit{IFix}\ f\ i) \\
\quad \mathit{ifold} \quad :: \mathit{IAlgebra}\ i\ f\ a \to \forall j.\mathit{IFix}\ f\ j \to a\ j
\end{array}
$$

Figure 3.12: Indexed Strictly-Positive Functor Class

$$
\begin{array}{l}
\textbf{type}\ \mathit{IAlgebra}\ i\ (f :: (i \to \mathit{Prop}) \to i \to \mathit{Prop})\ a = \\
\quad \forall (j :: i).f\ a\ j \to a\ j \\
\textbf{type}\ \mathit{IFix}_M\ i\ (f :: (i \to \mathit{Prop}) \to i \to \mathit{Prop})\ j = \\
\quad \forall a.\mathit{IAlgebra}\ i\ f\ a \to a\ j
\end{array}
$$

For type-soundness proofs we perform folds over proof-terms in order to establish propositions on the indices and hence make use of the fold operation provided by Church encodings. However, contrary to inductive datatypes we do not make use of propositions on proof-terms and hence do not need an induction principle for them. This also means that we do not need to keep track of the universal property of folds for proof-terms. Figure 3.12 defines the type class *ISPF* that collects the necessary reasoning interface for modularly building relations and indexed folds.

Since *Prop* is *impredicative* in Coq and induction-principles and universal properties are of no concern here, MTC's approach to modular inductive relations is sufficient for type-soundness proofs in Coq and we can universally instantiate *ISPF* with the definitions from MTC. However, other kinds of meta-theoretic proofs may require induction principles for proof terms and the approach is still limited to systems that support impredicativity.

Alternatively we can use a universe of indexed containers [Altenkirch and Morris, 2009] that does not have the above restrictions. An indexed container is essentially a container together with an assignment of indices for each shape and each position of that shape.

More formally, an i-indexed container $S \triangleright P \triangleright R$ is given by a family of shapes $S :: i \to *$ and family of position types $P :: (j :: i) \to S\ j \to *$ and

```
data ICont i where
  (_ ▷ _ ▷ _) :: (s :: i → ∗) →
                 (p :: ∀j.s j → ∗) →
                 (r :: ∀j s.p j s → i) → ICont i
ishape  (s ▷ p ▷ r) = s
ipos    (s ▷ p ▷ r) = p
irec    (s ▷ p ▷ r) = r
data IExt (c :: ICont i)
          (a :: i → Prop) (j :: i) :: Prop where
  IExt :: (s  :: ishape c j) →
          (pf :: ∀(p :: ipos c j s).a (irec c j s p)) →
          IExt c a j
data IW (c :: ICont i) (j :: i) :: Prop where
  ISup :: IExt c IW j → IW c j
```

Figure 3.13: Indexed Containers

an assignment $R :: (j :: i) \to (s :: S\ j) \to P\ j\ s \to i$ of indices for positions. Figure 3.13 gives the definition of the extension and the fixed point of an indexed container. Similarly to containers, one can generically define a fold operator for all indexed containers and construct the coproduct of two indexed containers.

Fixed points and fold operators can be defined generically on that universe similarly to Section 3.5.4. Indexed containers are also closed under coproducts and indexed algebras can be modularly composed using type classes.

3.6 Polynomial Functors

When choosing an approach to generic programming there is a trade-off between the expressivity of the approach, i.e. the collection of types it covers, and the functionality that can be implemented generically using the approach. The container universe is a very expressive universe in the sense that it supports a large class of types, but therefore the set of generic functions that can be implemented for containers is limited. In the previous section we have implemented generic functions for functorial mappings, fixed points, folds, induction and generic proofs about their properties for each container functor. Containers

```
class Eq a where
  eq       :: a → a → Bool
  eqTrue  :: ∀x y. eq x y = True  → xs = ys
  eqFalse :: ∀x y. eq x y = False → xs ≠ ys
```

Figure 3.14: Equality type class

are therefore well-suited as a solution for modularly defining datatypes and functions. But containers also include function types. Therefore any functionality that is not defined or decidable on function types cannot be implemented generically for every container. An example of such functionality is *equality*, which is in general not decidable for function types.

Other universes provide a different trade-off: they admit less types but allow more generic functionality. In this section we look at a universe that supports a generic implementation of equality testing and proofs about its correctness. Equality testing is used for example in the MTC framework in the implementation of a modular type-checker that tests if both branches of an **if** expression have the same type and that the function and argument type of a function application are compatible. Furthermore for reasoning about functions that use equality testing we need proofs of its correctness. For example, we want to prove that type-checked terms are indeed type-safe, i.e. they do not get stuck during evaluation. We thus include the equality function and the properties in an equality type class that is shown in Figure 3.14.

We choose the universe of univariate polynomial functors for the generic implementation of equality because it is well-studied, since it usually forms the basis of other datatype-generic programming approaches that take a *sums-of-products view* [Jansson and Jeuring, 1997] on datatypes. It can also be encoded in a lot of languages. For example the *regular* library [Van Noort et al., 2010; Yakushev et al., 2009] is an implementation in Haskell. Furthermore it is relatively easy to write instances for polynomial functors and a lot of signature functors that come up in practice are indeed polynomial functors.

Polynomial functors are a sub-class of container functors; we use this fact to integrate polynomial functors into our approach by writing a universe embedding into containers and allow mixing them freely with any container functors. Such universe embeddings have been studied by [Magalhães and Löh, 2012]. However, the universe of polynomial functors is not the only possible choice. There are universes of functors such as regular tree types [Morris et al., 2006]

```
data Poly = U | I | C Poly Poly | P Poly Poly
data Ext_P (c :: Poly) (a :: *) where
  EU :: Ext_P U a
  EI  :: a → Ext_P I a
  EL :: Ext_P c a → Ext_P (C c d) a
  ER :: Ext_P d a → Ext_P (C c d) a
  EP :: Ext_P c a → Ext_P d a → Ext_P (P c d) a
  class Polynomial f where
    pcode           :: Poly
    pto             :: Ext_P pcode a → f a
    pfrom           :: f a → Ext_P pcode a
    ptoFromInverse  :: ∀a.pto (pfrom a) = a
    pfromToInverse  :: ∀a.pfrom (pto a) = a
```

Figure 3.15: Polynomial Functors

or finite containers [Abbott et al., 2003] [1] that lie strictly between polynomial and container functors and also allow a generic implementation of equality.

Section 3.6.1 presents the definition of polynomial functors and Section 3.6.2 shows the embedding of polynomial functors into container functors. Generic equality for every fixed-point of a polynomial functors is defined in Section 3.6.3

3.6.1 Universe of Polynomial Functors

The codes *Poly* and interpretation Ext_P of the polynomial functor universe are shown in Figure 3.15. A polynomial functor is either the constant unit functor U, the identity functor I, a coproduct $C\ p_1\ p_2$ of two functors, or the cartesian product $P\ p_1\ p_2$ of two functors. The interpretation Ext_P is defined as an inductive family indexed by the codes. As before we define a type-class *Polynomial* that carries the conversion functions and isomorphism proofs. The definition of the class is also given in Figure 3.15.

[1] Also known as dependent polynomial functors [Gambino and Hyland, 2004] or shapely functors [Jaskelioff and Rypacek, 2012; Moggi et al., 1999].

Example As an example consider the functor *FunType* that can represent function types of an object language.

$$\textbf{data}\ \mathit{FunType}\ a = \mathit{TArrow}\ a\ a$$

It has a single constructor with two recursive positions for the domain and range types. Hence it can be represented by the code $P\ I\ I$. The conversion functions between the generic and conventional representation are given by

$$\begin{aligned}&\mathit{fromFunType} :: \mathit{FunType}\ a \to \mathit{Ext}_P\ (P\ I\ I)\ a\\ &\mathit{fromFunType}\ (\mathit{TArrow}\ x\ y) = \mathit{EP}\ (\mathit{EI}\ x)\ (\mathit{EI}\ y)\end{aligned}$$

$$\begin{aligned}&\mathit{toFunType} :: \mathit{Ext}_P\ (P\ I\ I)\ a \to \mathit{FunType}\ a\\ &\mathit{toFunType}\ (\mathit{EP}\ (\mathit{EI}\ x)\ (\mathit{EI}\ y)) = \mathit{TArrow}\ x\ y\end{aligned}$$

An instance for *FunType* is the following, with proofs omitted:

$$\begin{aligned}&\textbf{instance}\ \mathit{Polynomial}\ \mathit{FunType}\ \textbf{where}\\ &\quad \mathit{pcode} &&= P\ I\ I\\ &\quad \mathit{pto} &&= \mathit{toFunType}\\ &\quad \mathit{pfrom} &&= \mathit{fromFunType}\\ &\quad \mathit{ptoFromInverse} &&= ...\\ &\quad \mathit{pfromToInverse} &&= ...\end{aligned}$$

3.6.2 Universe Embedding

To write modular functions for polynomial functors we proceed in the same way as in Section 3.5 by showing that *Polynomial* is closed under coproducts and building the functionality of the *SPF* type class generically.

However, that would duplicate the generic functionality and would prevent us from using polynomial functors with containers. Since containers are closed under products and coproducts we can embed the universe of polynomial functors in the universe of containers. In order to do this, we have to derive a shape type from the code of a polynomial functor and a family of position types for each shape which are defined in Figure 3.16.

Shapes and Positions We compute the shape by recursing over the code. The constant unit functor and the identity functor have only one shape which is represented by a unit type. As in section 3.5 the shape of a coproduct is the

```
poly_S :: Poly → *
poly_S U         = ()
poly_S I         = ()
poly_S (C c d)   = poly_S c + poly_S d
poly_S (P c d)   = (poly_S c, poly_S d)
poly_P :: (c :: Poly) → poly_S c → *
poly_P U         ()          = Empty
poly_P I         ()          = ()
poly_P (C c d) (Left s)      = poly_P c s
poly_P (C c d) (Right s)     = poly_P d s
poly_P (P c d) (s1, s2)      =
  Either (poly_P c s1) (poly_P d s2)
```

Figure 3.16: Shapes and Positions of Polynomial Functors

coproduct of the shapes of the summands and the shape of a product is the product of shapes of the factors.

The definition of positions also proceeds by recursing over the code. The constant unit functor does not have any positions and the identity functor has exactly one position. For coproducts the positions are the same as the ones of the chosen summand and for a product we take the disjoint union of the positions of the shapes of the components.

Conversion The next essential piece for completing the universe embedding are conversions between the interpretations of the codes which are given in Figure 3.17. The function *ptoCont* converts the polynomial interpretation to the container interpretation. Similarly we define the function *pfromCont* that performs the conversion in the opposite direction. We omit the implementation.

To transport properties, like the correctness of equality in Figure 3.14, across these conversion functions we need to prove that they are inverses. These proofs proceed by inducting over the code; we omit them here.

Container Instance As the last step we derive an instance of *Container* from an instance of *Polynomial* in Figure 3.18. This way all the generic functionality of containers is also available for polynomial functors.

```
ptoCont :: (c :: Poly) → Ext_P c a → Ext (poly_S c ▷ poly_P c) a
ptoCont U       EU       = Ext () (λp → case p of )
ptoCont I       (EI a)   = Ext () (λ() → a)
ptoCont (C c d) (EL x)   = Ext (Left s) pf
  where Ext s pf = ptoCont c x
ptoCont (C c d) (ER y)   = Ext (Right s) pf
  where Ext s pf = ptoCont c y
ptoCont (P c d) (EP x y) = Ext (s1, s2) (λp → case p of
                                                Left p  → pf1 p
                                                Right p → pf2 p)
  where Ext s1 pf1 = ptoCont c x
        Ext s2 pf2 = ptoCont d y
pfromCont :: (c :: Poly) → Ext (poly_S c ▷ poly_P c) a → Ext_P c a
pfromCont = ...
```

Figure 3.17: Conversion between Polynomial Interpretations

3.6.3 Generic Equality

Performing the conversions between polynomial functors and containers in the definition of recursive functions makes it difficult to convince the termination checker to accept these definitions. So instead of using the generic fixed point provided by the container universe we define a generic fixed point on the polynomial functor universe directly.

```
data Fix_P (c :: Poly) = Fix_P (Ext_P c (Fix_P c))
```

We define the generic equality function *geq* mutually recursively with *go* that recurses over the codes and forms an equality function for a partially constructed fixed point.

```
geq :: (c :: Poly) → Fix_P c → Fix_P c → Bool
geq c (Fix_P x) (Fix_P y) = go c x y
  where
    go :: (d :: Poly) → Ext_P d (Fix_P c) → Ext_P d (Fix_P c) → Bool
    go U       EU       EU       = True
    go I       (EI x)   (EI y)   = geq x y
    go (C c d) (EL x)   (EL y)   = go c x y
```

```
instance Polynomial f ⇒ Container f where
  cont   = polyS pcode ▷ polyP pcode
  from   = ptoCont pcode ∘ pfrom
  to     = pto ∘ pfromCont pcode
  fromTo = ...
  toFrom = ...
```

Figure 3.18: Container Instance for Polynomial Functors

$$
\begin{array}{llll}
go\ (C\ c\ d)\ (EL\ x) & (ER\ y) & = False \\
go\ (C\ c\ d)\ (ER\ x) & (EL\ y) & = False \\
go\ (C\ c\ d)\ (ER\ x) & (ER\ y) & = go\ d\ x\ y \\
go\ (P\ c\ d)\ (EP\ x\ x')\ (EP\ y\ y') & & = go\ c\ x\ y \wedge go\ d\ x'\ y'
\end{array}
$$

In the same vein we can prove the properties of the *Eq* type class for this equality function using mutual induction over fixed points and partially constructed fixed points.

Of course $Fix_P\ c$ and $Fix_F\ (poly_S\ c \triangleright poly_P\ c)$ are isomorphic and we can transport functions and their properties across this isomorphism to get a generic equality function on the fixed-point defined by containers which can be used to instantiate the *Eq* type class in Figure 3.14.

```
instance Polynomial f ⇒ Eq (FixF f)
```

3.7 Case Study

As a demonstration of the advantages of our approach over MTC's Church encoding-based approach, we have ported the case study from [Delaware et al., 2013]. The study consists of soundness and continuity[2] proofs in addition to typing and evaluation functions of five reusable language features: 1) arithmetic expressions, 2) boolean expressions, 3) natural number pattern matching, 4) lambda abstraction and 5) a general recursion fixed-point operator.

Figure 3.19 presents the syntax of the expressions, values, and types provided by the features; each line is annotated with the feature that provides that set of definitions.

[2] of step-bounded evaluation functions

```
e ::= ℕ | e + e                                          Arith
    | 𝔹 | if e then e else e                             Bool
    | case e of { z ⇒ e ; S n ⇒ e }                      NatCase
    | lam x : T.e | e e | x                              Lambda
    | fix x : T.e                                        Recursion
V ::= ℕ                          Arith     T ::= nat          Arith
    | 𝔹                          Bool          | bool         Bool
    | closure e V̄                Lambda        | T → T        Lambda
```

Figure 3.19: mini-ML expressions, values, and types

In this section we discuss the benefits and trade-offs we have experienced while porting the case study to our approach.

Impredicative sets The higher-rank type in the definition of Fix_M

$$Fix_M\ (f :: Set \to Set) = \forall (a :: Set).Algebra\ f\ a \to a$$

causes $Fix_M\ f$ to be in a higher universe level than the domain of f. Hence to use $Fix_M\ f$ as a fixed-point of f we need an impredicative sort. MTC uses Coq's impredicative-set option for this which is known to be incompatible with axioms of classical logic.

When constructing the fixed-point of a container we do not need to raise the universe level and thus avoid impredicative sets.

Adequacy Adequacy of definitions is an important problem in mechanizations. One concern is the adequate encoding of fixed-points. MTC relies on a side-condition to show that for a given functor f the folding ($in_M :: f\ (Fix_M\ f) \to Fix_M\ f$) and unfolding ($out_M :: Fix_M\ f \to f\ (Fix_M\ f)$) are inverse operations, namely, that all appearing ($Fix_M\ f$) values need to have the universal property of folds. This side-condition raises the question if ($Fix_M\ f$) is an adequate fixed-point of f. The pairing of terms together with their proofs of the universal property do not form a proper fixed-point either, because of the possibility of different proof components for the same underlying terms.

Our approach addresses this adequacy issue: the *SPF* type class from Figure 3.1 requires that in_F and out_F are inverse operations without any side conditions on the values and containers give rise to proper *SPF* instances.

Module	Spec	Proof	Total	Description
FJ_tactics	193	99	292	Tactic library.
Functors	675	83	758	Functors, algebras and coproducts.
Containers	758	105	863	Universe of containers).
Polynomial	249	198	447	Universe of polynomial functors.
Equality	63	53	116	Equality for polynomial functors.
Total	1938	548	2476	

Table 3.1: Size statistic for the GDTC modular reasoning framework

Equality of terms Packaging universal properties with terms obfuscates equality of terms when using Church encodings. The proof component may differ for the same underlying term.

This shows up for example in type-soundness proofs in MTC. An extensible logical relation $\mathit{WTValue}\ (v, t)$ is used to represent well-typing of values. The judgement ranges over values and types. The universal properties are needed for inversion lemmas and thus the judgement needs to range over the variants that are packaged with the universal properties.

However, knowing that $\mathit{WTValue}\ (v, t)$ and $\mathit{proj1}\ t = \mathit{proj1}\ t'$ does not directly imply $\mathit{WTValue}\ (v, t')$, because of the possibly distinct proof component. To solve this situation auxiliary lemmas are needed that establish the implication. Other logical relations need similar lemmas. Every feature that introduces new rules to the judgments must also provide proof algebras for these lemmas.

In the case study two logical relations need this kind of auxiliary lemmas: the relation for well-typing and a sub-value relation for continuity. Both of these relations are indexed by two modular types and hence need two lemmas each. The proofs of these four lemmas, the declaration of abstract proof algebras and the use of the lemmas amounts to roughly 30 LoC per feature.

In our approach we never package proofs together with terms and hence this problem never appears. We thereby gain better readability of proofs and a small reduction in code size.

Code Size By moving to a datatype-generic approach the underlying framework for modular datatypes and modular relations and modular reasoning grew from about 1,200 LoC to about 2,500 LoC. Table 3.1 shows a detailed breakdown of the different modules which include both the universe of containers and polynomial functors and the generic implementations of fold, induction

	GDTC			MTC		
Module	**Spec**	**Proof**	**Total**	**Spec**	**Proof**	**Total**
Names	480	145	625	479	92	571
PNames	399	180	579	507	119	626
Total	879	325	1204	986	211	1197

Table 3.2: Size statistics of the type-safety infrastructure.

	GDTC			MTC		
Module	**Spec**	**Proof**	**Total**	**Spec**	**Proof**	**Total**
Arith	415	150	565	522	492	1014
Bool	441	151	592	560	169	729
Lambda	779	171	950	1223	159	1382
Mu	401	31	432	481	26	507
NatCase	212	25	237	282	12	294
ArithLambda	59	9	68	103	7	110
BoolLambda	60	11	71	112	7	119
Total	2367	548	2915	3283	872	4155

Table 3.3: Size statistics of the feature mechanizations.

and equality.

The feature-independent infrastructure for type-safety is defined in modules *Names* and *PNames*. These contain declarations of modular functions for typing and evaluation, and modular proofs of continuity and type safety of evaluation. Table 3.2 contains a breakdown of these modules and a comparison with the original versions from the MTC case study. The GDTC versions avoid some duplicate declarations for decorated and undecorated fixed-points and save code by eliminating the need to reason about universal properties. However the GDTC approach also requires instance declarations for containers, in particular indexed container instances for predicates on environments, which are new obligations and nullify any savings. Both the MTC and the GDTC version are about 1,200 LoC.

The size of the implementation of the five modular feature components and feature interactions is roughly 830 LoC per feature in the original MTC case study. By switching from Church encodings to a datatype-generic approach we

	GDTC			MTC		
Composition	**Spec**	**Proof**	**Total**	**Spec**	**Proof**	**Total**
A	52	3	55	47	3	50
AL	76	5	81	49	9	58
BL	76	5	81	57	9	66
AB	79	3	82	73	6	79
ABL	103	5	108	129	9	138
MiniML	91	5	96	67	10	77
Total	477	26	503	422	46	468

Table 3.4: Size statistics of the language compositions

stripped away on average about 230 LoC of additional proof obligations needed for reasoning with Church encodings per feature. However, instantiating the MTC interface amounts to roughly 40 LoC per feature while our approach requires about 70 LoC per feature for the container and polynomial instances.

By using the generic equality and generic proofs about its properties we can remove the feature-specific implementations from the case study. This is about 40 LoC per feature. In total we have reduced the average size of a feature implementation by about 240 LoC to 590 LoC. Table 3.3 shows a detailed breakdown of the different features implemented as part of the case study.

The last piece of the case study consists of the language compositions: a subset of features is chosen to form a language and the type-safety theorem is derived for that language from the modular proof algebras. The breakdown for six different compositions is shown in Table 3.4. The GDTC compositions derive local container instances for composed functors to avoid costly and repetitive type class resolutions. This puts the GDTC variants slightly above the MTC version in terms of code size.

Summary The case study shows that our approach can effectively replace the MTC approach and offers simplifications for programming and reasoning about modular datatypes and relations. Another benefit is the applicability in proof-assistants that do not offer impredicative sorts to implement the MTC approach.

In terms of development effort the savings achieved by switching to the container based approach and removing boilerplate functions like equality testing

are in the order of a 25% code size reduction per feature. Since the user does not need to concern herself with the preservation of the universal properties of folds in her definitions, our approach offers a less complex framework that can result in less development effort not only in terms of code size, but also in terms of coding time and mental effort.

3.8 Related and Future Work

There is a fairly large amount of related work on modular programming and datatype-generic proramming. Below we discuss the most relevant related to this chapter: work on modular proofs and datatype-generic programming in proof assistants.

DGP in proof-assistants Datatype-generic programming started out as a form of language extension such as PolyP [Jansson and Jeuring, 1997] or Generic Haskell [Löh et al., 2003]. Yet Haskell has turned out to be powerful enough to implement datatype-generic programming in the language itself and over the time a vast number of DGP libraries for Haskell have been proposed [Cheney and Hinze, 2002; Lämmel and Jones, 2003; Oliveira et al., 2006; Yakushev et al., 2009; Chakravarty et al., 2009; Mitchell and Runciman, 2007; Magalhães et al., 2010]. Compared with a language extension, a library is much easier to develop and maintain.

There are multiple proposals for performing datatype-generic programming in proof-assistants using the flexibility of dependent-types [Verbruggen et al., 2008; Altenkirch and McBride, 2003; Benke et al., 2003; Löh and Magalhães, 2011; Altenkirch and Morris, 2009]. This setting not only allows the implementation of generic functions, but also of generic proofs. The approaches vary in terms of how strictly they follow the positivity or termination restrictions imposed by the proof-assistant. Some circumvent the type-checker at various points to simplify the development or presentation while others put more effort in convincing the type-checker and termination checker of the validity [Morris et al., 2006]. However, in any of the proposals there does not seem to be any fundamental problem caused by the positivity or termination restrictions.

DGP for modular proofs Modularly composing semantics and proofs about the semantics has also been addressed by [Schwaab and Siek, 2013] in the context of programming language meta-theory. They perform their development in Agda and, similar to our approach, they also use a universe approach based on polynomial functors to represent modular datatypes. They

split relations for small-step operational semantics and well-typing on a feature basis. However, the final fixed-points are constructed manually instead of having a generic representation of inductive families.

Schwaab and Siek did not model functors, folds or induction operators concretely but instead rely also on manual composition of algebras. Therefore, their definitions have a more natural directly-recursive style. But as a consequence, some of their definitions are not structurally recursive. Unfortunately Schwaab and Siek circumvent Agda's termination checker instead of stratifying their definitions.

Using Coq's type classes both MTC and our approach provide more automation in the final composition of datatypes, functions and proofs. Agda features instance arguments that can be used to replace type classes in various cases. Schwaab and Siek [2013]'s developments took place when Agda's implementation did not perform recursive resolution, and as a result Agda did not support automation of the composition to the extent that is needed for DTC-like approaches. However, as of Agda version 2.4.2 instance argument resolution is recursive. Hence it should now be possible to augment Schwaab and Siek's approach with full automation for composition and also port our approach to Agda.

A notable difference is that Schwaab and Siek do not define a dependent recursor for induction but instead completely rely on non-dependent recursion over relations. Therefore the needs for a strong generic induction principle does not arise in their work and MTC's generic folds for relations is sufficient for their approach. However, MTC's Church encodings are rejected by Agda's type-checker because Agda is a fully predicative system.

Combining different DGP approaches We have shown an embedding of the universe of polynomial functors into the universe of containers. Similar inclusions between universes have been presentend in the literature [Morris et al., 2007]. Magalhães and Löh [2012] have ported several popular DGP approaches from Haskell to Agda and performed a formal comparison by proving inclusion relations between the approaches including a port of the *regular* Haskell library that is equivalent to our polynomial functor universe. However, they did not consider containers in their comparison.

DGP approaches differ in terms of the class of datatypes they capture and the set of generic functions that can be implemented for them. Generic functions can be transported from a universe into a sub-universe. However, we are not aware of any DGP library with a systematic treatment of universes where each generic function is defined at the most general universe that supports

that function.

Automatic derivation of instances Most, if not all, generic programming libraries in Haskell use Template Haskell to derive the generic representation of user-defined types and to derive the conversion functions between them.

The GMeta [Lee et al., 2012] framework includes a standalone tool that also performs this derivation for Coq. Similarly, deriving instances for *Container* and *Polynomial* classes automatically is possible. An alternative is presented in [Chapman et al., 2010]. Chapman et al.'s goal is to reflect the datatype declarations of the programming language automatically in the language itself, which are then immediately available for datatype-generic programming.

3.9 Scientific Output

This chapter is based on the contents of the article:

> Keuchel, S., & Schrijvers, T. (2013). Generic Datatypes la Carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, WGP 13, pages 13-24. ACM.

The main contributions of this work are

- Bringing existing work from the fields of modularity, genericity and type theory together in the same framework.
- A novel representation of proof algebras that serves as a connection between these fields.

Below we make the border between prior work and this work precise.

Firstly, both modular programming and datatype-generic programming have been indepedently well-studied.

- Modular programming and the expression problem have been studied extensively. The Datatypes à la Carte [Swierstra, 2008] approach is an existing solution in a purely functional programming setting. Metatheory à la Carte [Delaware et al., 2013] extends DTC to modular proving. We reuse most of the definitions from MTC, in particular automation of compositions. The main difference and contribution of this work is the change to a new datatype-generic based representation of signature functors that provides an alternative to the Church encodings of MTC. This alternative representation addresses multiple shortcoming of MTC.

- Datatype-generic programming (DGP) or polytypic programming [Jansson and Jeuring, 1997] is an established field in functional programming which has also seen extensive use in dependently-type theory [Benke et al., 2003]. The universe of containers is a well-studied subject, including generic functionality like generic induction for containers [Abbott et al., 2003]. The container-based representation comes with generic fixed-points, folds and induction principles that meet the requirements of proof-assistants.

The novelty of our setting is that we combine modular programming with the container presentation.

Secondly, we present a generic representation of proof algebras that is independent of the particular generic universe. All-modalities have been implicitly used before by [Benke et al., 2003] to define generic induction, but specialized for a particular universe. [Morris, 2007] models all-modalities explicitly for datatype-generic universes but does not use them for generic induction principles. Our contribution is to provide the last missing piece to define induction independently of the particular generic universe used: formulate uniform proof-algebras on explicit all-modalities.

Chapter 4

Modular Monadic Effects

Traditional proofs by *structural induction* are not modular. The MTC and GDTC frameworks open up the recursion in these proofs to allow for extensibility. However, the closedness of inductive proofs is not the only concern that inhibits modularity in proofs. For instance, in the metatheory of programming languages with side-effects, existing non-modular formalizations typically assume a concrete set of effects that the feature of a particular language at hand is using. The *semantic functions*, *theorem statements* (such as *type safety*) and corresponding *proofs* of these formalizations have the concrete set of effects hardwired.

No Effect Modularity The statement of type soundness for a language depends intimately on the effects it uses, making it particularly difficult to achieve modularity. Consider defining two features in MTC: mutable references Ref_F and errors Err_F. Both of these introduce an effect to any language, the former state and the latter the possibility of raising an error. These effects show up in the type of their evaluation algebras:

$$eval_{Ref} :: Algebra_M\ Ref_F\ (Env \to (Value, Env))$$
$$eval_{Err} :: Algebra_M\ Err_F\ (Maybe\ Value)$$

MTC supports the composition of two algebras over different functors as long as they have the same carrier. That is not the case here, making the two algebras incompatible. This problem can be solved by anticipating both effects in both algebras by choosing a common and uniform carrier for both algebras:

$$eval_{Ref} :: Algebra_M\ Ref_F\ (Env \to (Maybe\ Value, Env))$$
$$eval_{Err} :: Algebra_M\ Err_F\ (Env \to (Maybe\ Value, Env))$$

This anticipation is problematic for modularity: the algebra for references mentions the effect of errors even though it does not involve them, while a language that includes references does not necessarily feature errors. More importantly, the two algebras cannot be composed with a third feature that introduces yet another effect (e.g., local environments) without further foresight. It is impossible to know in advance all the effects that new features may introduce.

Hence, a key challenge in modularizing effects is how to generalize every definition – *semantic functions*, *theorem statements* and *proofs* – to a form that is *uniform* and *general* enough to account for any desired set of *potential effects* instead of hardwiring one specific set of effects.

This chapter introduces the 3MT approach to tackle the problem of modular type-safety proofs for effectful languages. We first give a high-level characterization of how the approach achieves *uniformity* and thus *modularity* before giving an outline of this chapter.

Modular Semantic Functions Fortunately, for semantic functions there is already a good solution: *monads* and *monad transformers*. Monads are a well-established mechanism for defining the semantics of languages with effects. Moreover, monads give a *uniform representation* for effectful computations independent of the effects which is an important point for modularization. With the help of monad transformers, monads can be modularly composed.

For each effect, a monad subclass captures a set of primitive operations. These abstract type classes form the main interface that is used for implementing and reasoning about features and their effects, instead of using a particular monad (or monad stack) directly. This ensures that definitions are general enough without assuming a concrete implementation. Reasoning about monadic programs is commonly performed using *equational reasoning* [Gibbons and Hinze, 2011; Oliveira et al., 2010] which is a prevalent techniques in functional programming. For each of the monad type-classes, a set of algebraic laws governs the interaction between primitive operations.

Modular Soundness Proofs To solve the key challenge of modularizing and reusing theorems and proofs of type soundness, we split the classic type soundness theorems into three separate parts:

1. Reusable *feature theorems* capture the essence of type soundness for an

individual feature. They depend on that feature's syntax, typing relation, semantic function and only the effects used therein. At the same time, they abstract over the syntax, semantics and effects of other features. This means that the addition of new features with other types of effects *does not affect* the existing feature theorem proofs.

To achieve the abstraction over other effects, a feature uses a polymorphic monad that is constrained by monad sub-classes. As a consequence, it only establishes the well-typing of the resulting denotations with respect to the effects of the declared subclass constraints.

2. Reusable *effect theorems* fix the monad of denotations and consequently the set of effects. They take well-typing proofs of monadic denotations expressed in terms of a constrained polymorphic monad and which mention only a subset of effects, and turn them into well-typings with respect to a fixed monad and all the effects it provides.

   Effect theorems reason fully at the level of denotations and abstract over the details of language features like syntax and semantic functions. Consequently the same effect theorem will work for any languages that use the particular combination of effects captured by the theorem.

3. Finally, *language theorems* establish type soundness for a particular language. They require no more effort than to instantiate the set of features and the set of effects (i.e., the monad), thus tying together the respective feature and effect theorems into an overall proof.

Outline Monads form the underlying mechanism to define the semantics of languages denotationally and monad transformers are used to modularize the semantics. Section 4.1 introduces the 3MT monad library that we use throughout this chapter.

Section 4.2 presents a monadic and uniform representation of algebras of semantic function to define denotational semantics with effects without fixing the particular set of effects. It further combines the *modular datatypes* of MTC/GDTC and *monads/monad transformers* to define semantic function algebras on a per-feature basis, and thus make the denotational semantics also independent of a particular set of language features.

Section 4.3 examines *feature theorems* and Section 4.4 concerns itself with *effect theorems* and *language theorems*. Section 4.5 discusses our case study of 5 features with their feature theorems, 8 different effect theorems and 28 fully mechanized languages, including a mini-ML variant with errors and references.

4.1 The 3MT Monad Library

3MT includes a monad library to support effectful semantic functions using *monads* and *monad transformers*, and provides *algebraic laws* for reasoning. Monads provide a uniform representation for encapsulating computational effects such as mutable state, exception handling, and non-determinism. Monad transformers allow monads supporting the desired set of effects to be built. Algebraic laws are the key to *modular* reasoning about monadic definitions.

3MT implements the necessary definitions of *monads* and *monad transformers* as a Coq library inspired by the Haskell *monad transformer library* (MTL) [Liang et al., 1995]. Our library refines the MTL in two key ways in order to support modular reasoning using algebraic laws:

1. While algebraic laws can only be documented informally in Haskell, our library fully integrates them into type class definitions using Coq's expressive type system.

2. 3MT systematically includes laws for all monad subclasses, several of which have not been covered in the functional programming literature before.

4.1.1 Monad Classes

Figure 4.1 summarizes the library's key classes, definitions and laws. The type class *Monad* describes the basic interface of monads.[1] The type $m\ a$ denotes computations of type m which produce values of type a when executed. The function *return* lifts a value of type a into a (pure) computation that simply produces the value. The *bind* function $\gg\!=$ composes a computation $m\ a$ producing values of type a, with a function that accepts a value of type a and returns a computation of type $m\ b$. The function $\gg$ defines a special case of bind that discards the intermediate value:

$$(\gg) :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$$
$$ma \gg mb = ma \gg\!= \backslash\_ \to mb$$

The **do** notation is syntactic sugar for the *bind* operator: **do** $\{x \leftarrow f; g\}$ means $f \gg\!= \lambda x \to g$.

[1] In contrast to pending changes to Haskell's standard library, we do not require *Applicative* but merely *Functor* as a superclass of *Monad* (cf. `https://wiki.haskell.org/Functor-Applicative-Monad_Proposal`. We do however exclude the *fail* method (cf. `https://wiki.haskell.org/MonadFail_Proposal`).

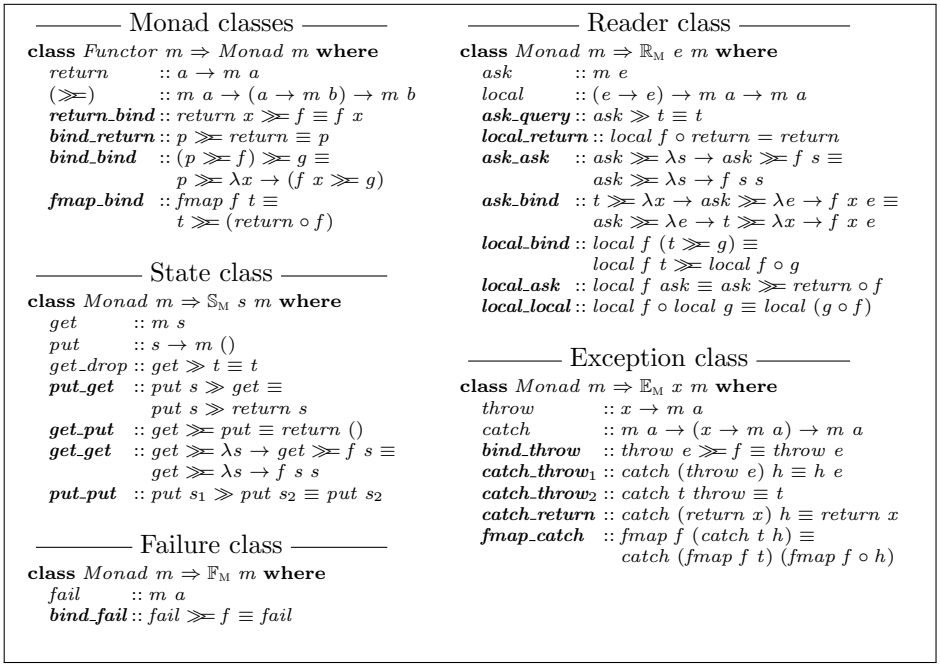


Figure 4.1: Key classes, definitions and laws from 3MT's monadic library.

The primitive operations of each effect are defined in *monad subclasses* (denoted by subscript M) such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$. For example, *get* is a method of the $\mathbb{S}_{\mathrm{M}}$ class to retrieve the state without changing it.

4.1.2 Algebraic Laws

Each monad (sub)class includes a set of algebraic laws that govern its operations. These laws are an integral part of the definition of the monad type classes and constrain the possible implementations to sensible ones. Thus, even without knowing the particular implementation of a type class, we can still modularly reason about its behavior via these laws. This is crucial for supporting modular reasoning [Oliveira et al., 2010].

The first three laws for the *Monad* class are standard, while the last law (***fmap_bind***) relates *fmap* and *bind* in the usual way. Each monad subclass also includes its own set of laws. The laws for various subclasses can be found scattered throughout the functional programming literature, such as for

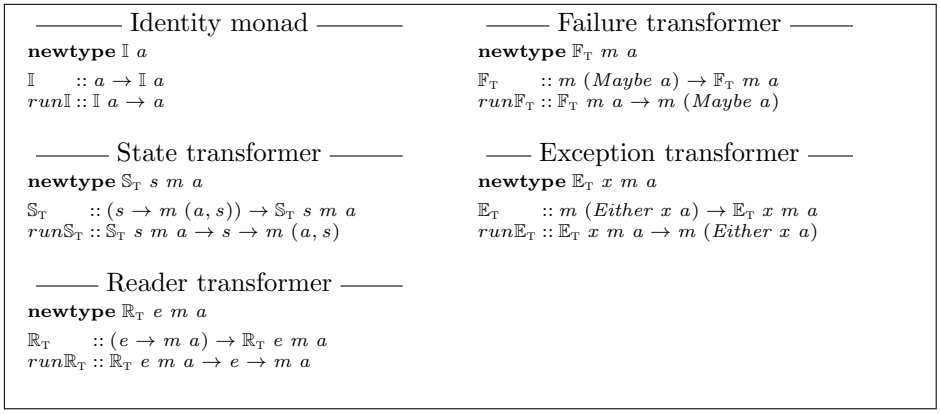


Figure 4.2: Monad transformers

failure [Gibbons and Hinze, 2011] and state [Gibbons and Hinze, 2011; Oliveira et al., 2010]. Yet, as far as we know, 3MT is the first to systematically bring them together. Furthermore, although most laws have been presented in the semantics literature in one form or another, we have not seen some of the laws in the functional programming literature. One such example are the laws for the exception class:

- The ***bind_throw*** law generalizes the ***bind_fail*** law: a sequential computation is aborted by throwing an exception.
- The ***catch_throw***$_1$ law states that the exception handler is invoked when an exception is thrown in a *catch*.
- The ***catch_throw***$_2$ law indicates that an exception handler is redundant if it just re-throws the exception.
- The ***catch_return*** law states that a *catch* around a pure computation is redundant.
- The ***fmap_catch*** law states that pure functions (*fmap* f) distribute on the right with *catch*.

4.1.3 Monad Transformers

Particular monads can be built from basic monad types such as the identity monad ($\mathbb{I}$) and monad transformers including the failure ($\mathbb{F}_T$), mutable state

Arithmetic Expressions	$Monad\ m$
Boolean Expressions	$Monad\ m$
Errors	$\mathbb{E}_{\mathrm{M}}\ ()\ m$
References	$\mathbb{S}_{\mathrm{M}}\ Store\ m$
Lambda	$\mathbb{R}_{\mathrm{M}}\ Env\ m, \mathbb{F}_{\mathrm{M}}\ m$

Figure 4.3: Effects used by the case study's evaluation algebras.

($\mathbb{S}_{\mathrm{T}}$), reader ($\mathbb{R}_{\mathrm{T}}$), and exception ($\mathbb{E}_{\mathrm{T}}$) transformers that are shown in Figure 4.2. These transformers are combined into different monad stacks with ($\mathbb{I}$) at the bottom. Constructor and extractor functions such as ($\mathbb{S}_{\mathrm{T}}$) and ($run\mathbb{S}_{\mathrm{T}}$) provide the signatures of the functions for building and running particular monads/transformers.

4.1.4 Discussion

Our monad library contains a number of other classes, definitions and laws apart from the definitions discussed here. This includes infrastructure for other types of effects (e.g. writer effects), as well as other infrastructure from the MTL. There are roughly 30 algebraic laws in total.

4.2 Modular Monadic Semantics

Features can utilize the monad library included with 3MT to construct algebras for semantic functions. In order to support extensible effects, a feature needs to abstract over the monad implementation used. Any implementation which includes the required operations is valid. These operations are captured in type classes and each class offers a set of primitive operations. The abstract type classes form the main interface that is used for implementing and reasoning about features and their effects, instead of using a particular monad (or monad stack) directly. This ensures that definitions are general enough without assuming a concrete implementation.

Take for example monadic evaluation algebras for references and error:

$$eval_{Ref} :: \mathbb{S}_{\mathrm{M}}\ Store\ m \Rightarrow Algebra_M\ Ref_F\ (m\ a)$$
$$eval_{Err} :: \mathbb{E}_{\mathrm{M}}\ ()\ m \Rightarrow Algebra_M\ Err_F\ (m\ a)$$

These algebras use monad subclasses such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$ to *constrain* the monad required by the feature, allowing the monad to have more effects than

```
— Simplified value interface —
type Value

loc    :: Int → Value
stuck  :: Value
unit   :: Value
isLoc  :: Value → Maybe Int

——— Expression functor ———
data Ref_F a = Ref a
             | DeRef a
             | Assign a a

type Store = [Value]

—— Monadic typing algebra ——
typeof_Ref :: 𝔽_M m ⇒
              Algebra_M Ref_F (m Type)
typeof_Ref rec (Ref e) = do
  t ← rec e
  return (tRef t)
typeof_Ref rec (DeRef e) = do
  te ← rec e
  maybe fail return (isTRef te)
typeof_Ref rec (Assign e1 e2) = do
  t1 ← rec e1
  case isTRef t1 of
    Nothing → fail
    Just t  → do
      t2 ← rec e2
      if (t ≡ t2)
        then return tUnit
        else fail

—— Simplified type interface ——
type Type

tRef    :: Type → Type
tUnit   :: Type
isTRef  :: Type → Maybe Type

– Monadic evaluation algebra –
eval_Ref :: 𝕊_M Store m ⇒
            Algebra_M Ref_F (m Value)
eval_Ref rec (Ref e) = do
  v   ← rec e
  env ← get
  put (v : env)
  return (loc (length env))
eval_Ref rec (DeRef e) = do
  v   ← rec e
  env ← get
  return $
    case isLoc v of
      Nothing → stuck
      Just n  →
        maybe stuck id (fetch n env)
eval_Ref rec (Assign e1 e2) = do
  v   ← rec e1
  env ← get
  case isLoc v of
    Nothing → return stuck
    Just n  → do
      v2 ← rec e2
      put (replace n v2 env)
      return unit
```

Figure 4.4: Syntax and type definitions for references.

those used in the feature. These two algebras can be combined to create a new evaluation algebra with type:

$$(\mathbb{S}_{\mathrm{M}}\ m\ s, \mathbb{E}_{\mathrm{M}}\ m\ x) \Rightarrow \mathit{Algebra}_M\ (\mathit{Ref}_F \oplus \mathit{Err}_F)\ (m\ a)$$

The combination imposes both type class constraints while the monad type remains extensible with new effects. The complete set of effects used by the evaluation functions for the five language features used in our case study of Section 4.5 are given in Figure 4.3.

4.2.1 Example: References

Figure 4.4 illustrates this approach with definitions for the signature functor for expressions and the evaluation and typing algebras for the reference feature. Other features have similar definitions.

For the sake of presentation the definitions are slightly simplified from the actual ones in Coq. For instance, we have omitted issues related to the extensibility of the syntax for values (*Value*) and types (*Type*). *Value* and *Type* are treated as abstract datatypes with a number of smart constructor functions (c.f. Section 2.2.2): *loc*, *stuck*, *unit*, *tRef* and *tUnit* denote respectively reference locations, stuck values, unit values, reference types and unit types. There are also matching functions *isLoc* and *isTRef* for checking whether a term is a location value or a reference type, respectively.

The type Ref_F is the functor for expressions of references. It has constructors for creating references (*Ref*), dereferencing (*DeRef*) and assigning (*Assign*) references. The evaluation algebra $eval_{Ref}$ uses the state monad for its reference environment, which is captured in the type class constraint $\mathbb{S}_{\mathrm{M}}$ *Store m*. The typing algebra ($typeof_{Ref}$) is also monadic, using the failure monad to denote ill-typing.

4.2.2 Effect-Dependent Theorems

Monadic semantic function algebras are compatible with new effects and algebraic laws facilitate writing extensible proofs over these monadic algebras. Effects introduce further challenges to proof reuse, however: each combination of effects induces its own type soundness statement. Consider the theorem LSound$_S$for a language with references which features a store σ and a store typing Σ that are related through the store typing judgement $\Sigma \vdash \sigma$:

$$\forall e, t, \Sigma, \sigma. \left\{ \begin{array}{c} typeof\ e \equiv return\ t \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow$$
$$\exists v, \Sigma', \sigma'. \left\{ \begin{array}{c} put\ \sigma \gg [\![e]\!] \equiv put\ \sigma' \gg return\ v \\ \Sigma' \supseteq \Sigma \\ \Sigma' \vdash v : t \\ \Sigma' \vdash \sigma' \end{array} \right\} \qquad (\text{LSound}_S)$$

The initial store σ is well-formed w.r.t. the initial store typing Σ, the final store typing Σ' is an extension of the initial one, and the final store σ' is well-formed w.r.t. Σ'. The *put* operation is used to constrain the initial and final store of the monadic computation.

Contrast this with the theorem LSound$_E$ for a language with errors, which must account for the computation possibly ending in an exception being thrown which is modeled by a disjunction in the conclusion:

$$\forall e, t. typeof\ e \equiv return\ t \rightarrow$$
$$(\exists v. [\![e]\!] \equiv return\ v \wedge \vdash v : t) \vee (\exists x. [\![e]\!] \equiv throw\ x) \qquad (\text{LSound}_E)$$

Clearly, the available effects are essential for the formulation of the theorem. A larger language which involves both exceptions and state requires yet another theorem LSOUND$_{ES}$ where the impact of both effects cross-cut one another[2]:

$$\forall e, t, \Sigma, \sigma. \left\{ \begin{array}{c} \mathit{typeof}\ e \equiv \mathit{return}\ t \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow$$
$$\exists v, \Sigma', \sigma'. \left\{ \begin{array}{c} \mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{put}\ \sigma' \gg \mathit{return}\ v \\ \Sigma' \supseteq \Sigma \\ \Sigma' \vdash v : t \\ \Sigma' \vdash \sigma' \end{array} \right\}$$
$$\vee$$
$$\exists x. \mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{throw}\ x \qquad (\text{LSOUND}_{ES})$$

Modular formulations of LSOUND$_S$ and LSOUND$_E$ are useless for proving a modular variant of LSOUND$_{ES}$, because their induction hypotheses have the wrong form. The hypothesis for LSOUND$_E$ requires the result to be of the form *return* v, disallowing *put* $\sigma' \gg$ *return* v (the form required by LSOUND$_S$). Similarly, the hypothesis for LSOUND$_S$ does not account for exceptions occurring in subterms. In general, without anticipating additional effects, type soundness theorems with fixed sets of effects cannot be reused modularly.

4.3 Monadic Type Safety

As demonstrated in Section 4.2.2, soundness theorem which hardwire the set of effects limit their modularity by rendering proofs for soundness with distinct effects incompatible. In this Section we discuss how to split type soundness into smaller theorems and how to generalize the parts for modularization.

4.3.1 Three-Step Approach

In order to preserve a measure of modularity, we do not prove type soundness directly for a given feature, but by means of more generic theorems. In order to maximize compatibility, the statement of the type soundness theorem of a feature cannot hardwire the set of effects. This statement must instead rephrase type soundness in a way that can adapt to any superset of a feature's effects. Hence, similar to the modularization of the semantics in Section 4.2, the idea is to abstract over any monad that provides the effects of a feature.

[2] A similar proliferation of soundness theorems can be found in TAPL [Pierce, 2002].

Our approach to state and prove type soundness of features, is to establish that the monadic evaluation and typing algebras of a feature satisfy an extensible well-typing relation of computations, defined in terms of *effect-specific typing rules*. This relation forms the basis of a *feature theorem* that has *uniform shape* independent of specific features or effects (except for constraints on the monad). A feature's proof algebra for the feature theorem only uses the typing rules required for the effects specific to that feature. The final language combines the typing rules of all the language's effects into a closed relation and the *feature theorem* for the complete set of features into a type soundness theorem of the whole language.

As discussed in Section 4.2.2 the type soundness statement of a language still depends on the set of effects of all the features of a language. The statement is however independent of the set of features. This allows us to split the type soundness theorem further into a reusable *effect theorem* and a *language theorem*. In summary, we split the type soundness into three kinds of theorems:

- FSound: a reusable *feature theorem* that is only aware of the effects that a feature uses,
- ESound: an *effect theorem* for a fixed set of known effects, and
- LSound: a *language theorem* which combines the two to prove soundness for a specific language.

Figure 4.5 illustrates how these reusable pieces fit together to build a proof of soundness. Each feature provides a proof algebra for FSound which relies on the typing rules (WFM-X) for the effects it uses. Each unique statement of soundness for a combination of effects requires a new proof of ESound. The proof of LSound for a particular language is synthesized entirely from a single proof of ESound and a combination of proof algebras for FSound.

Note that there are several dimensions of modularity here. A feature's proof of FSound only depends on the typing rules for the effects that feature uses and can thus be used in any language which includes those typing rules. The typing rules themselves can be reused by any number of different features. ESound depends solely on a specific combination of effects and can be reused in any language which supports that unique combination, e.g. both LSound_R and LSound_{AR} use ESound_S.

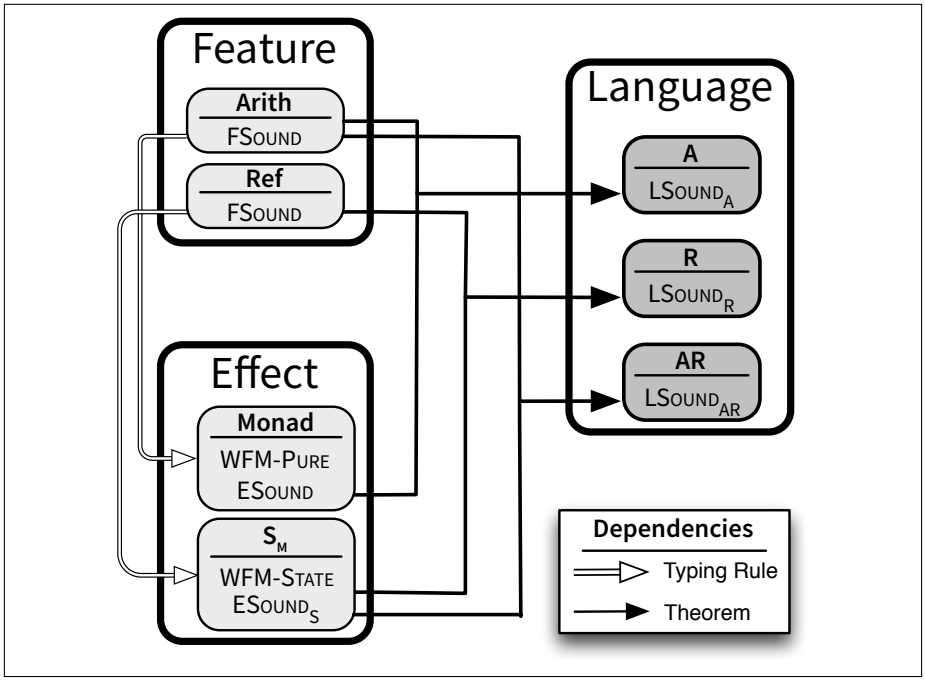


Figure 4.5: Dependency Graph

4.3.2 Typing of Monadic Computations

For type-soundness we make use of two separate typing relations: a relation for values and one for computations. The value typing relation

$$\Sigma \vdash v : t$$

is implicitly parameterized by an environment type *env* and has three indices: an environment Σ, a value v and a type t. The computation typing relation has the form:

$$\Sigma \vdash_M v_m : t_m$$

The relation is polymorphic in an environment type *env* and an evaluation monad type m. The parameters Σ, v_m and t_m have types *env*, (m *Value*) and (*Maybe Type*) respectively.

The extensible feature theorem FSound states that $[\![\cdot]\!]$ and *typeof* are related by the typing relation:

$$\frac{}{\Sigma \vdash_M v_m \ : \ \mathit{fail}} \ \textsc{Wfm-Illtyped}$$

$$\frac{\Sigma \vdash v \ : \ t}{\Sigma \vdash_M \mathit{return}\ v \ : \ \mathit{return}\ t} \ \textsc{Wfm-Return}$$

Figure 4.6: Typing rules for pure monadic values.

$$\forall e, \Sigma. \quad \Sigma \vdash_M [\![e]\!] : \mathit{typeof}\ e \qquad (\textsc{FSound})$$

The modular typing rules for this relation can impose constraints on the environment type *env* and monad type m. A particular language must instantiate *env* and m in a way that satisfies all the constraints imposed by the typing rules used in its features.

Figure 4.6 lists the two base typing rules of this relation. These do not constrain the evaluation monad and environment types and are the only rules needed for pure features. The (WFM-Illtyped) rule denotes that nothing can be said about computations (v_m) which are ill-typed. The (WFM-Return) rule ensures that well-typed computations only yield values of the expected type.

4.3.3 Monolithic Soundness for a Pure Feature

To see how the reusable theorem works for a pure feature, consider the proof of soundness for the boolean feature.

Proof Using the two rules of Figure 4.6, we can show that FSound holds for the boolean feature. The proof has two cases. The boolean literal case is handled by a trivial application of (WFM-Return). The second case, for

conditionals, is more interesting[3]:

$$(\vdash_M [\![e_c]\!] : \mathit{typeof}\ e_c) \to (\vdash_M [\![e_t]\!] : \mathit{typeof}\ e_t) \to (\vdash_M [\![e_e]\!] : \mathit{typeof}\ e_e) \to$$

$$\vdash_M \left(\begin{array}{l} \textbf{do} \\ \quad v \leftarrow [\![e_c]\!] \\ \quad \textbf{case}\ \mathit{isBool}\ v\ \textbf{of} \\ \quad\quad \mathit{Just}\ b \quad \to \\ \quad\quad\quad \textbf{if}\ b\ \textbf{then}\ [\![e_t]\!] \\ \quad\quad\quad\quad \textbf{else}\ [\![e_e]\!] \\ \quad\quad \mathit{Nothing} \to \mathit{stuck} \end{array}\right) : \left(\begin{array}{l} \textbf{do} \\ \quad t_c \leftarrow \mathit{typeof}\ e_c \\ \quad t_t \leftarrow \mathit{typeof}\ e_t \\ \quad t_e \leftarrow \mathit{typeof}\ e_e \\ \quad \mathit{guard}\ (\mathit{isTBool}\ t_c) \\ \quad \mathit{guard}\ (\mathit{eqT}\ t_t\ t_e) \\ \quad \mathit{return}\ t_t \end{array}\right) \tag{WFM-If-Vc}$$

Because $[\![\cdot]\!]$ and *typeof* are polymorphic in the monad, we cannot directly inspect the values they produce. We can, however, perform case analysis on the derivations of the proofs produced by the induction hypothesis that the subexpressions are well-formed, $\vdash_M [\![e_c]\!] : \mathit{typeof}\ e_c$, $\vdash_M [\![e_t]\!] : \mathit{typeof}\ e_t$, and $\vdash_M [\![e_e]\!] : \mathit{typeof}\ e_e$. The final rule used in each derivation determines the shape of the monadic value produced by $[\![\cdot]\!]$ and *typeof*. Assuming that only the pure typing rules of Figure 4.6 are used for the derivations, we can divide the proof into two cases depending on whether e_c, e_t, or e_e was typed with (WFM-Illtyped).

- If any of the three derivations uses (WFM-Illtyped), the result of *typeof* is *fail*. Hence (WFM-Illtyped) resolves the case.
- Otherwise, each of the subderivations was built with (WFM-Return) and the evaluation and typing expressions can be simplified using the ***return_bind*** monad law.

$$\vdash_M \left(\begin{array}{l} \textbf{case}\ \mathit{isBool}\ v_c\ \textbf{of} \\ \quad \mathit{Just}\ b \to \\ \quad\quad \textbf{if}\ b\ \textbf{then}\ \mathit{return}\ v_t \\ \quad\quad\quad \textbf{else}\ \mathit{return}\ v_e \\ \quad \mathit{Nothing} \to \mathit{stuck} \end{array}\right) : \left(\begin{array}{l} \textbf{do} \\ \quad \mathit{guard}\ (\mathit{isTBool}\ t_c) \\ \quad \mathit{guard}\ (\mathit{eqT}\ t_t\ t_e) \\ \quad \mathit{return}\ t_t \end{array}\right)$$

[3]We omit the environment Σ to avoid clutter.

After simplification, the typing expression has replaced the bind with explicit values which can be reasoned with. If $(isTBool\ t_c)$ is *false*, then the typing expression reduces to *fail* and well-formedness again follows from the WFM-Illtyped rule. Otherwise ($t_c \equiv TBool$), and we can apply the canonical forms lemma

$$\vdash v : TBool \rightarrow \exists b.isBool\ v \equiv Just\ b$$

to establish that v_c is of the form $(Just\ b)$, reducing the evaluation to either $(return\ v_e)$ or $(return\ v_t)$. A similar case analysis on $(eqT\ t_t\ t_e)$ will either produce *fail* or $(return\ t_t)$. The former is trivially true, and both $\vdash_M return\ v_t : return\ t_t$ and $\vdash_M return\ v_e : return\ t_t$ hold in the latter case from the induction hypotheses.

4.3.4 Modular Sublemmas

The above proof assumed that only the pure typing rules of Figure 4.6 were used to type the subexpressions of the if-expression, which is clearly not the case when the boolean feature is included in an effectful language. Instead, case analyses are performed on the extensible typing relation in order to make the boolean feature theorem compatible with new effects. Case analyses over the extensible $\vdash_M$ relation are accomplished using extensible proof algebras which are folded over the derivations provided by the induction hypothesis, as outlined in Section 3.5.7.

In order for the boolean feature's proof of FSound to be compatible with a new effect, each extensible case analysis requires a proof algebra for the new typing rules the effect introduces to the $\vdash_M$ relation. More concretely, the conditional case of the previous proof can be dispatched by folding a proof algebra for the property WFM-If-Vc over $\vdash_M [\![v_c]\!] : typeof\ t_c$. However, each new effect induces a new case for this proof algebra.

These proof algebras are examples of *interactions* [Batory et al., 2011] from the setting of modular component-based frameworks. In essence, an interaction is functionality (e.g., a function or a proof) that is only necessary when two components are combined. Our case is an interaction between the boolean feature and any effect.

Importantly, these proof algebras do not need to be provided up front when developing the boolean algebra, but can instead be modularly resolved by later by separate proof algebras for the interaction of the boolean feature and each effect.

Nevertheless, the formulation of the properties proved by extensible case analysis has an impact on modularity. WFM-If-Vc is specific to the proof of

$$\frac{\Sigma \vdash_M v_m : t_m \quad (\forall v\ t\ \Sigma'.(\Sigma' \supseteq \Sigma) \to (\Sigma' \vdash v : t) \to (\Sigma' \vdash_M k_v\ v : k_t\ t))}{\Sigma \vdash_M (v_m \gg\!= k_v) : (t_m \gg\!= k_t)}\ \text{WFM-BIND}$$

Figure 4.7: Reusable sublemma for monadic binds.

FSOUND in the boolean feature; proofs of FSOUND for other features require different properties and thus different proof algebras. Relying on such specific properties can lead to a proliferation of proof obligations for each new effect.

4.3.5 Reusable Bind Sublemma

Alternatively, the boolean feature can use a proof algebra for a stronger property that is also applicable in other proofs, cutting down on the number of feature interactions. One such stronger, more general sublemma WFM-BIND is shown in Figure 4.7. It relates the monadic bind operation to well-typing. If the two subcomputations v_m and t_m on the left-hand side of the bind yield values v and t, then invoking the continuations k_v and k_t with these values gives rise to well-typed computations. The rule requires this for any possible well-typed combination of v and t.

A proof of WFM-IF-VC follows from two applications of this stronger property. The advantage of WFM-BIND is clear: it can be reused to deal with case analyses in other proofs of FSOUND, while a proof of WFM-IF-VC has only a single use. As WFM-BIND is a desirable property for typing rules, the case study focuses on that approach.

4.4 Effect and Language Theorems

The second phase of showing type soundness is the *effect theorem*, that proves a statement of soundness for a fixed set of effects.

4.4.1 Pure Languages

For pure effects, the soundness statement is straightforward:

$$\forall v_m\ t.\ \vdash_M v_m : \mathit{return}\ t \Rightarrow \exists v.v_m \equiv \mathit{return}\ v \wedge \vdash v : t \qquad (\text{ESOUND}_P)$$

$$\frac{}{\Sigma \vdash_M \mathit{throw}\ x\ :\ t_m}\ \textsc{Wfm-Throw}$$

$$\frac{\Sigma \vdash_M m \gg\!= k\ :\ t_m \qquad \forall\ \Sigma' \supseteq \Sigma\ x\ .\ \Sigma' \vdash_M h\ x \gg\!= k\ :\ t_m}{\Sigma \vdash_M \mathit{catch}\ m\ h \gg\!= k\ :\ t_m}\ \textsc{Wfm-Catch}$$

Figure 4.8: Typing rules for exceptional monadic values.

Each effect theorem is proved by induction over the derivation of $\vdash_M v_m : \mathit{return}\ t$. ESound$_P$ fixes the irrelevant environment type to the unit type () and the evaluation monad to the pure monad $\mathbb{I}$. Since the evaluation monad is fixed, the proof of ESound$_P$ only needs to consider the pure typing rules of Figure 4.6. The proof of the effect theorem is straightforward: Wfm-Illtyped could not have been used to derive $\vdash_M v_m : \mathit{return}\ t$, and Wfm-Return provides both a witness for v and a proof that it is of type t.

The statement of soundness for a pure language built from a particular set of features is similar to ESound$_P$:

$$\forall e, t.\mathit{typeof}\ e \equiv \mathit{return}\ t \Rightarrow \exists v.[\![e]\!] \equiv \mathit{return}\ v \wedge \vdash v : t \qquad (\text{LSound}_P)$$

The proof of LSound$_P$ is an immediate consequence of the reusable proofs of FSound and ESound$_P$. Folding a proof algebra for FSound over e provides a proof of $\vdash_M [\![e]\!] : \mathit{return}\ t$, satisfying the first assumption of ESound$_P$. LSound$_P$ follows immediately.

4.4.2 Errors

The evaluation algebra of the error language feature uses the side effects of the exception monad, requiring new typing rules.

Typing Rules Figure 4.8 lists the typing rules for monadic computations involving exceptions which are parameterized by a type x for exceptional values. Wfm-Throw states that $(\mathit{throw}\ x)$ is typeable with any type. Wfm-Catch states that binding the results of both branches of a *catch* statement will produce a monad with the same type. While it may seem odd that this rule is formulated in terms of a continuation $\gg\!= k$, it is essential for compatibility with the proofs algebras required by other features. As described in Section 4.3.2, extensible proof algebras over the typing derivation will now need cases for the

two new rules. To illustrate this, consider the proof algebra for the general purpose WFM-Bind property. This algebra requires a proof of:

$$(\Sigma \vdash_M \mathit{catch}\ e\ h \gg\!= k\ :\ t_m) \to$$
$$(\forall v\ t\ \Sigma' \supseteq \Sigma.\ (\Sigma' \vdash\ v\ :\ t) \to \Sigma' \vdash_M\ k_v\ v\ :\ k_t\ t) \to$$
$$\Sigma \vdash_M (\mathit{catch}\ e\ h \gg\!= k) \gg\!= k_v : t_m \gg\!= k_t$$

With the continuation, we can first apply the associativity law to reorder the binds so that WFM-Catch can be applied:

$$(\mathit{catch}\ e\ h \gg\!= k) \gg\!= k_v = \mathit{catch}\ e\ h \gg\!= (k \gg\!= k_v).$$

The two premises of the rule follow immediately from the inductive hypothesis of the lemma, finishing the proof. Without the continuation, the proof statement only binds *catch e h* to v_m, leaving no applicable typing rules.

Effect Theorem The effect theorem, ESound$_E$, for a language whose only effect is exceptions reflects that the evaluation function is either a well-typed value or an exception.

$$\forall v_m\ t. \vdash_M v_m : \mathit{return}\ t \to$$
$$\exists x. v_m \equiv \mathit{throw}\ x \vee \exists v. v_m \equiv \mathit{return}\ v \wedge \vdash v : t \qquad (\text{ESound}_E)$$

The proof of ESound$_E$ is again by induction on the derivation of $\vdash_M v_m$: *return t*. The irrelevant environment can be fixed to the unit type (), while the evaluation monad is the exception monad $\mathbb{E}_{\mathrm{T}}\ x\ \mathbb{I}$.

The typing derivation is built from four rules: the two pure rules from Figure 4.6 and the two exception rules from Figure 4.8. The case for the two pure rules is effectively the same as before, and WFM-Throw is straightforward. In the remaining case, ($v_m \equiv \mathit{catch}\ e'\ h$), and we can leverage the fact that the evaluation monad is fixed to conclude that either ($\exists v. e' \equiv \mathit{return}\ v$) or ($\exists x. e' \equiv \mathit{throw}\ x$). In the former case, *catch e′ h* can be reduced using ***catch_return***, and the latter case is simplified using ***catch_throw***$_1$. In both cases, the conclusion then follows immediately from the assumptions of WFM-Catch. The proof of the language theorem LSound$_E$is similar to LSound$_P$ and is easily built from ESound$_E$ and FSound.

4.4.3 References

Typing Rules Figure 4.9 lists the two typing rules for stateful computations. To understand the formulation of these rules, consider LSound$_S$, the

$$\frac{\forall \sigma, \Sigma \vdash \sigma \;\rightarrow\; \Sigma \vdash_M k\ \sigma : t_m}{\Sigma \vdash_M \; get \gg\!= k \;:\; t_m}\ \textsc{Wfm-Get}$$

$$\frac{\Sigma' \vdash \sigma \qquad \Sigma' \supseteq \Sigma \qquad \Sigma' \vdash_M k : t_m}{\Sigma \vdash_M \; put\ \sigma \gg k \;:\; t_m}\ \textsc{Wfm-Put}$$

Figure 4.9: Typing rules for stateful monadic values.

statement of soundness for a language with a stateful evaluation function. The statement accounts for both the typing environment Σ and evaluation environment σ by imposing the invariant that σ is well-formed with respect to Σ. FSound however, has no such conditions (which would be anti-modular in any case). We avoid this problem by accounting for the invariant in the typing rules themselves:

- WFM-Get requires that the continuation k of a *get* is well-typed under the invariant.
- WFM-Put requires that any newly installed environment maintains this invariant.

The intuition behind these premises is that effect theorems will maintain these invariants in order to apply the rules.

Effect Theorem The effect theorem for mutable state proceeds again by induction over the typing derivation. The evaluation monad is fixed to $\mathbb{S}_{\mathrm{T}}\ \Sigma\ \mathbb{I}$ and the environment type is fixed to $[\,\mathit{Type}\,]$ with the obvious definitions for $\supseteq$.

- The proof case for the two pure rules is again straightforward.
- For WFM-Get we have that $put\ \sigma \gg [\![e]\!] \equiv put\ \sigma \gg get \gg\!= k$. After reducing this to $(k\ \sigma)$ with the ***put_get*** law, the result follows immediately from the rule's assumptions.
- Similarly, for WFM-Put we have that $put\ \sigma \gg [\![e]\!] \equiv put\ \sigma \gg put\ \sigma' \gg k$. After reducing this to $put\ \sigma' \gg k$ with the ***put_put*** law, the result again follows immediately from the rule's assumptions.

$$\frac{\forall \gamma.\ \Gamma \vdash \gamma \ \rightarrow\ \Gamma \vdash_M k\ \gamma : t_m}{\Gamma \vdash_M \ \mathit{ask} \gg\!= k\ :\ t_m}\ \textsc{Wfm-Ask}$$

$$\frac{\begin{array}{c}\forall\ \gamma.\ \Gamma \vdash \gamma \rightarrow \Gamma' \vdash f\ \gamma \qquad \Gamma' \vdash_M m\ :\ \mathit{return}\ \ t'_m \\ \forall v.\ \vdash v\ :\ t'_m \rightarrow \Gamma \vdash_M (k\ v) : t_m\end{array}}{\Gamma \vdash_M \ \mathit{local}\ f\ m \gg\!= k\ :\ t_m}\ \textsc{Wfm-Local}$$

$$\frac{}{\Gamma \vdash_M \bot\ :\ t_m}\ \textsc{Wfm-Bot}$$

Figure 4.10: Typing rules for environment and failure monads.

4.4.4 Lambda

The case study represents the binders of the lambda feature using PHOAS [Chlipala, 2008] to avoid many of the boilerplate definitions and proofs about term well-formedness found in first-order representations.

The Environment Effect 3MT neatly hides the variable environment of the evaluation function with a reader monad $\mathbb{R}_{\mathrm{M}}$, unlike MTC which passes the environment explicitly. This new effect introduces the two new typing rules listed in Figure 4.10. Unsurprisingly, these typing rule are similar to those of Figure 4.9. The rule for *ask* is essentially the same as WFM-Get. The typing rule for *local* differs slightly from WFM-Put. Its first premise ensures that whenever f is applied to an environment that is well-formed in the original typing environment Γ, the resulting environment is well-formed in some new environment Γ'. The second premise ensures the body of *local* is well-formed in this environment according to some type T, and the final premise ensures that k is well-formed when applied to any value of type T. The intuition behind binding the *local* expression in some k is the same as with *put*.

The Partiality Effect The lambda feature also introduces the possibility of non-termination to the evaluation function, which is disallowed by Coq. MTC solves this problem by combining *mixin algebras* with a bounded fixed-point function. This function applies an algebra a bounded number of times, returning a $\bot$ value when the bound is exceeded. Because MTC represented $\bot$ as a value, all evaluation algebras needed to account for it explicitly. In the monadic setting, 3MT elegantly represents $\bot$ with the *fail* primitive of the

failure monad. This allows terminating features to be completely oblivious to whether a bounded or standard fold is used for the evaluation function, resulting in a much cleaner semantics. WFM-Bot allows $\bot$ to have any type.

4.4.5 Modular Effect Compositions

As we have seen, laws are essential for proofs of FSound. The proofs so far have involved only up to one effect and the laws regulate the behavior of that effect's primitive operations.

Languages often involve more than one effect, however. As outline in Section 4.2.2 the effect theorem depends on the set of available effects, and in the case of multiple effects the theorem is different from the theorem for any proper subset of effects. Moreover, the proofs of effect theorems must reason about the interaction between multiple effects. There is a trade-off between fully instantiating the monad for the language as we have done previously, and continuing to reason about a constrained polymorphic monad. The former is easy for reasoning, while the latter allows the same language proof to be instantiated with different implementations of the monad. In the latter case, additional *effect interaction* laws are required.

The following sections discuss compositions of different set of effects, their effect theorem statement, and the necessary interaction laws to prove the effect theorem.

4.4.6 State and Exceptions

Consider the effect theorem which fixes the evaluation monad to support exceptions and state. The statement of the theorem mentions both kinds of effects by requiring the evaluation function to be run with a well-formed state σ and by concluding that well-typed expressions either throw an exception or return a value. The WFM-Catch case this theorem has the following goal:

$$
\begin{gathered}
(\Sigma \vdash \sigma \ : \ \Sigma) \\
\rightarrow \\
\exists\, \Sigma', \sigma', v. \left\{ \begin{array}{c} \mathit{put}\ \sigma \gg \mathit{catch}\ e\ h \gg\!= k \equiv \mathit{put}\ \sigma' \gg \mathit{return}\ v \\ \Sigma' \vdash v : t \end{array} \right\} \\
\vee \\
\exists\, \Sigma', \sigma', x. \left\{ \begin{array}{c} \mathit{put}\ \sigma \gg \mathit{catch}\ e\ h \gg\!= k \equiv \mathit{put}\ \sigma' \gg \mathit{throw}\ x \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \end{array} \right\}
\end{gathered}
$$

$$\forall \Sigma, \Gamma, \delta, \gamma, \sigma, e_E, e_T. \left\{ \begin{array}{c} \gamma, \delta \vdash \ e_E \equiv e_T \\ \Sigma \vdash \sigma \ : \ \Sigma \\ \Sigma \vdash \gamma \ : \ \Gamma \\ \mathit{typeof}\ e_T \equiv \mathit{return}\ t \end{array} \right\} \rightarrow$$

$$\exists\ \Sigma', \sigma', v. \left\{ \begin{array}{c} \mathit{local}\ (\lambda\_.\gamma)\ (\mathit{put}\ \sigma \gg [\![e]\!]_E) \\ \equiv \mathit{local}\ (\lambda\_.\gamma)\ (\mathit{put}\ \sigma' \gg \mathit{return}\ v) \\ \Sigma' \vdash v : t \end{array} \right\} \vee$$

$$\exists\ \Sigma', \sigma', v. \left\{ \begin{array}{c} \mathit{local}\ (\lambda\_.\gamma)\ (\mathit{put}\ \sigma \gg [\![e]\!]_E) \\ \equiv \mathit{local}\ (\lambda\_.\gamma(\mathit{put}\ \sigma' \gg \bot) \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\} \vee \qquad (\text{ESOUND}_{ESRF})$$

$$\exists\ \Sigma', \sigma', v. \left\{ \begin{array}{c} \mathit{local}\ (\lambda\_.\gamma)\ (\mathit{put}\ \sigma \gg [\![e]\!]_E) \\ \equiv \mathit{local}\ (\lambda\_.\Gamma(\mathit{put}\ \sigma' \gg \mathit{throw}\ t) \\ \Sigma' \vdash \sigma' \ : \ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\}$$

Figure 4.11: Effect theorem statement for languages with errors, state, an environment and failure.

In order to apply the induction hypothesis to e and h, we need to precede them by a $(\mathit{put}\ \sigma)$. Hence, $(\mathit{put}\ \sigma)$ must be pushed under the *catch* statement through the use of a law governing the behavior of *put* and *catch*. There are different choices for this law, depending on the monad that implements both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$. We consider two reasonable choices, based on the monad transformer compositions $\mathbb{E}_{\mathrm{T}}\ x\ (\mathbb{S}_{\mathrm{T}}\ s\ \mathbb{I})$ and $\mathbb{S}_{\mathrm{T}}\ s\ (\mathbb{E}_{\mathrm{T}}\ x\ \mathbb{I})$:

- Either *catch* passes the current state into the handler:

$$\mathit{put}\ \sigma \gg \mathit{catch}\ e\ h \equiv \mathit{catch}\ (\mathit{put}\ \sigma \gg e)\ h$$

- Or *catch* runs the handler with the initial state:

$$\mathit{put}\ \sigma \gg \mathit{catch}\ e\ h \equiv \mathit{catch}\ (\mathit{put}\ \sigma \gg e)\ (\mathit{put}\ \sigma \gg h)$$

The WFM-CATCH case is provable under either choice. As the ESOUND_{ES} proof is written as an extensible theorem, the two cases are written as two separate proof algebras, each with a different assumption about the behavior of the interaction. Since the cases for the other rules are impervious to the choice, they can be reused with either proof of WFM-CATCH.

Exceptional Environment

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{R}_{\mathrm{M}}\ m) \Rightarrow \mathbb{ER}_{\mathrm{M}}\ x\ g\ m \textbf{ where}$$
$$\quad \textit{local\_throw} :: \textit{local}\ f\ (\textit{throw}\ e) \equiv \textit{throw}\ e$$
$$\quad \textit{local\_catch} :: \textit{local}\ f\ (\textit{catch}\ e\ h) \equiv$$
$$\qquad \textit{catch}\ (\textit{local}\ f\ e)\ (\lambda x.\textit{local}\ f\ (h\ x))$$

Exceptional Failure

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{F}_{\mathrm{M}}\ m) \Rightarrow \mathbb{FS}_{\mathrm{M}}\ x\ m \textbf{ where}$$
$$\quad \textit{catch\_fail} :: \textit{catch}\ \textit{fail}\ h \equiv \textit{fail}$$
$$\quad \textit{fail\_neq\_throw} :: \textit{fail} \neq \textit{throw}\ x$$

Exceptional State Failure

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{S}_{\mathrm{M}}\ s\ m, \mathbb{F}_{\mathrm{M}}\ m) \Rightarrow \mathbb{EFS}_{\mathrm{M}}\ x\ m \textbf{ where}$$
$$\quad \textit{put\_fail\_throw} :: \textit{put}\ \sigma \gg \textit{fail} \neq \textit{put}\ \sigma' \gg \textit{throw}\ x$$

Exceptional State

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{F}_{\mathrm{M}}\ m) \Rightarrow \mathbb{ES}_{\mathrm{M}}\ x\ m \textbf{ where}$$
$$\quad \textit{put\_ret\_throw} :: \textit{put}\ \sigma \gg \textit{return}\ a \neq \textit{put}\ \sigma' \gg \textit{throw}\ x$$
$$\quad \textit{put\_throw} :: \forall A\ B.\textit{put}\ \sigma \gg \textit{throw\_}A\ x \equiv \textit{put}\ \sigma' \gg \textit{throw\_}A\ x \rightarrow$$
$$\qquad \textit{put}\ \sigma \gg \textit{throw\_}B\ x \equiv \textit{put}\ \sigma' \gg \textit{throw\_}B\ x$$

Alternate Exceptional State laws

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{F}_{\mathrm{M}}\ m) \Rightarrow \mathbb{ES}_{\mathrm{M}_1}\ x\ m \textbf{ where}$$
$$\quad \textit{put\_catch}_1 :: \textit{put}\ \sigma \gg \textit{catch}\ e\ h \equiv \textit{catch}\ (\textit{put}\ \sigma \gg e)\ h$$

Or

$$\textbf{class } (\mathbb{E}_{\mathrm{M}}\ x\ m, \mathbb{F}_{\mathrm{M}}\ m) \Rightarrow \mathbb{ES}_{\mathrm{M}_2}\ x\ m \textbf{ where}$$
$$\quad \textit{put\_catch}_2 :: \textit{put}\ \textbf{env} \gg \textit{catch}\ e\ h \equiv$$
$$\qquad \textit{catch}\ (\textit{put}\ \sigma \gg e)\ (\lambda x \rightarrow \textit{put}\ \sigma \gg h\ x)$$

Figure 4.12: Interaction laws

4.4.7 State, Reader and Exceptions

A language with references, errors and lambda abstractions features four effects: state, exceptions, an environment and failure. The language theorem for such a language relies on the effect theorem $\textsc{Esound}_{ESRF}$ given in Figure 4.11. The proof of $\textsc{Esound}_{ESRF}$ is similar to the previous effect theorem proofs, and makes use of the full set of interaction laws given in Figure 4.12. Perhaps the most interesting observation here is that because the environment monad only makes local changes, we can avoid having to choose between laws regarding how it interacts with exceptions. Also note that since we are representing

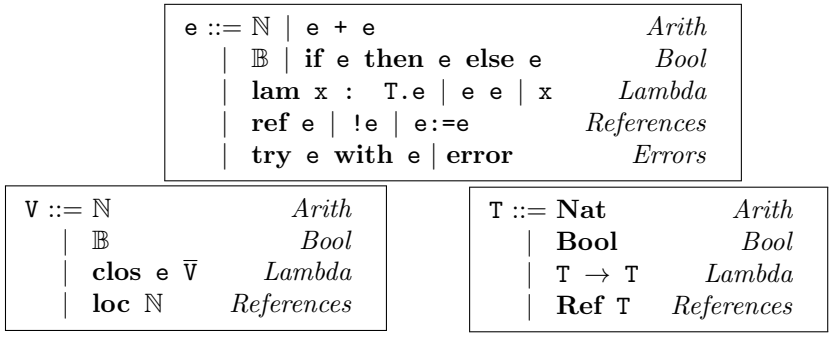


Figure 4.13: mini-ML expressions, values, and types

Module	Spec	Proof	Total	Description
MTC	803	388	1191	MTC Framework.
MonadLib	1350	201	1551	3MT monad and monad transfomers.
Names	358	94	452	Type-safety infrastructure.
PNames	229	100	329	PHOAS type-safety infrastructure.
Total	2740	783	3523	

Table 4.1: Size statistic for the 3MT framework for modular effect reasoning

nontermination using a failure monad $\mathbb{F}_{\mathrm{M}}\ m$, the ***catch_fail*** law conforms to our desired semantics.

4.5 Case Study

As a demonstration of the 3MT framework, we have built a set of five reusable language features and combined them to build a family of languages which includes a mini-ML [Clément et al., 1986] variant with references and errors. The study includes pure boolean and arithmetic features as well as effectful features for references, errors and lambda abstractions.

The study builds twenty eight different combinations of the features which are all possible combinations with at least one feature providing values. Figure 4.13 presents the syntax of the expressions, values, and types provided; each line is annotated with the feature that provides that set of definitions.

Table 4.1 gives an overview of the size in *lines of code* (LoC) of different

Module	Spec	Proof	Total
Pure	79	60	139
Fail	65	11	76
Except	97	59	156
Reader	95	24	119
State	107	20	127
Total	483	174	657

Table 4.2: Size statistics of the monadic value typing relations.

parts of the 3MT library. The 3MT library consists of about 3,550 LoC of which 1,200 LoC comprise the MTC implementation of modular datatypes and modular induction, 1,550 LoC comprise the implementation of the monad transformers and their algebraic laws and finally the infrastructure for monadic type safety consists of 800 LoC. All of this code is not specific to language features and is therefore reusable.

A breakdown of the size of the effect implementations is given in Table 4.2. These include the modular typing rules for the specified effect and a proof algebra for the reusable bind lemma of these rules.

The size in LoC of the implementation of semantic evaluation and typing functions and the reusable feature theorem for each language feature is given in Table 4.3. Two kinds of feature interactions appear in the case study.

- The PHOAS representation of binders requires an auxiliary equivalence relation, the details of which are covered in the MTC paper [Delaware et al., 2013]. The soundness proofs of language theorems for languages which include binders proceed by induction over this equivalence relation instead of expressions. The reusable feature theorems of other features need to be lifted to this equivalence relation. In our case study only the *Lambda* feature includes binders. The equivalence relations and liftings of the other features are contained in the modules *ArithEqv*, *BoolEqv*, *ErrorEqv* and *RefEqv* which are also listed in Table 4.3. Together, they represent about 16% of the LoC of the feature implementations. However, this code does not represent feature interactions with the *Lambda* feature, but rather with variable binding and is reusable in combination with other features that use binding.

- Inversion lemmas for the well-formed value relation, such as in the proof of FSound for the boolean feature in Section 4.3.2, are proven by in-

Module	Spec	Proof	Total
Arith	524	110	634
Bool	590	122	712
Error	348	60	408
Ref	895	229	1124
Lambda	840	173	1013
ArithEqv	158	13	171
BoolEqv	174	14	188
ErrorEqv	145	14	159
RefEqv	229	16	245
Lambda_Arith	40	16	56
Lambda_Bool	40	16	56
Lambda_Ref	53	35	88
Total	4036	818	4854

Table 4.3: Size statistics of the feature implementations.

duction over the relation. Inversion lemmas are needed for natural numbers, boolean, store locations and closures. Except for closures, these inversion lemmas are dispatched by tactics hooked into the type class inference algorithm. For the closure inversion, manually written proof algebras are used. These are implemented in the modules *Lambda_Arith*, *Lambda_Bool* and *Lambda_Ref* which together form amount to 200 LoC or about 4% of the feature specific code and are thus negligible.

Table 4.4 lists the sizes of the effect theorems for each set of effects used in the case study. The letters in the module name encode the effect set: $P = Pure$ (no effects), $E = Exceptions$, $R = Reader$, $F = Fail$ and $S = State$. The effect theorems are completely un-modular, but are reusable for languages that use the specific set of effects. With an average of 220 LoC, these effect theorems are much smaller than the modular feature implementations.

Each language needs on average 90 LoC to assemble its semantic functions and soundness proofs from those of its features and the effect theorem for its set of effects. Table 4.5 contains a detailed overview. The letters encode the set of used features: $A = Arith, B = Bool, E = Error, L = Lambda$ and $R = References$.

Apart from the language specific compositions, our approach significantly

Module	Spec	Proof	Total
ESoundP	59	20	79
ESoundE	76	61	137
ESoundRF	95	50	145
ESoundS	87	70	157
ESoundERF	210	31	241
ESoundRFS	179	104	283
ESoundES	134	164	298
ESoundERFS	199	233	432
Total	1039	733	1772

Table 4.4: Size statistics of the effect theorems.

reduces the amount of un-reusable code. The code with the least reuse are the effect theorems. However, for any language composition, the size of the effect theorem is overshadowed by the size of the feature implementations.

4.6 Related Work

While previous work has explored the basic techniques of modularizing dynamic semantics of languages with effects, our work is the first to show how to also do modular proofs. Adding the ability to do modular proofs required the development of novel techniques for reasoning about modular components with effects.

4.6.1 Functional Models for Modular Side Effects

Monads and Monad Transformers Since Moggi [1989] first proposed monads to model side-effects, and Wadler [1992] popularized them in the context of Haskell, various researchers (e.g., [Jones and Duponcheel, 1993; Steele, 1994]) have sought to modularize monads. Monad transformers emerged [Cenciarelli and Moggi, 1993; Liang et al., 1995] from this process, and in later years various alternative implementation designs facilitating monad (transformer) implementations, have been developed, including Filinski [1999]'s layered monads and Jaskelioff [2011]'s Monatron.

Module	Spec	Proof	Total	Module	Spec	Proof	Total
A	25	4	29	*BL*	58	9	67
B	26	10	36	*BARE*	58	19	77
AB	27	12	39	*BAL*	64	9	73
R	35	9	44	*LR*	75	18	93
AR	37	12	49	*ARL*	80	18	98
BR	37	12	49	*BRL*	81	18	99
AE	37	22	59	*ALE*	85	18	103
BE	37	22	59	*BARL*	86	16	102
BAR	38	14	52	*BLE*	87	18	105
ABE	38	25	63	*BALE*	93	18	111
RE	53	18	71	*LRE*	106	9	116
ARE	56	17	73	*ARLE*	113	9	122
BRE	57	17	74	*BRLE*	115	9	124
AL	58	9	67	*BARLE*	121	9	130
Total	1783	400	2184				

Table 4.5: Size statistics of the language compositions.

Monads and Subtyping Filinski's MultiMonadic MetaLanguage (M^3L) [Filinski, 2007, 2010] embraces the monadic approach, but uses subtyping (or subeffecting) to combine the effects of different components. The subtyping relation is fixed at the program or language level, which does not provide the adaptability we achieve with constrained polymorphism.

Algebraic Effects and Effect Handlers In the semantics community the algebraic theory of computational effects [Plotkin and Power, 2002] has been an active area of research. Many of the laws about effects, which we have not seen before in the context of functional programming, can be found throughout the semantics literature. Our first four laws for exceptions, for example, have been presented by Levy [2006].

A more recent model of side effects are effect handlers. They were introduced by Plotkin and Pretnar [2009] as a generalization from exception handlers to handlers for a range of computational effects, such as I/O, state, and nondeterminism. Bauer and Pretnar [2015] built the language *Eff* around effect handlers and show how to implement a wide range of effects in it. Kammar et al. [2012] showed that effect handlers can be implemented in terms of delimited continuations or free monads.

The major advantage of effect handlers over monads is that they are more easily composed, as any composition of effect operations and corresponding handlers is valid. In contrast, not every composition of monads is a monad. In the future, we plan on investigating the use of effect handlers instead of monad transformers, which could potentially reduce the amount of work involved on proofs about interactions of effects.

Other Effect Models Other useful models have been proposed, such as *applicative functors* [Mcbride and Paterson, 2008] and *arrows* [Hughes, 2000], each with their own axioms and modularity properties.

4.6.2 Modular Effectful Semantics

There are several works on how to modularize semantics with effects, although none of these works considers reasoning.

Mosses [2004] modularizes structural operational semantics by means of a label transition system where extensible labels capture effects like state and abrupt termination. Swierstra [2008] presents modular syntax with functor coproducts and modular semantics with algebra compositions. To support effects, he uses modular syntax to define a free monad. The effectful semantics for this free monad is not given in a modular manner, however. Jaskelioff et al. [2011] present a modular approach for operational semantics on top of Swierstra's modular syntax, although they do not cover conventional semantics with side-effects. Both Schrijvers and Oliveira [2010] and Bahr and Hvitved [2011] have shown how to define modular semantics with monads for effects; this is essentially the approach followed in this paper for modular semantics.

4.6.3 Effects and Reasoning

Non-Modular Monadic Reasoning Although monads are a purely functional way to encapsulate computational-effects, programs using monads are challenging to reason about. The main issue is that monads provide an abstraction over purely functional models of effects, allowing functional programmers to write programs in terms of abstract operations like $\gg=$, *return*, or *get* and *put*. One way to reason about monadic programs is to remove the abstraction provided by such operations [Hutton and Fulger, 2008]. However, this approach is fundamentally non-modular.

Modular Monadic Reasoning Several more modular approaches to modular monadic reasoning have been pursued in the past.

One approach to modular monadic reasoning is to exploit *parametricity* [Reynolds, 1983; Wadler, 1989]. Voigtländer [2009] has shown how to derive parametricity theorems for type constructor classes such as *Monad*. Unfortunately, the reasoning power of parametricity is limited, and parametricity is not supported by proof-assistants like Coq.

A second technique uses *algebraic laws*. Liang and Hudak [1996] present one of the earliest examples of using algebraic laws for reasoning. They use algebraic laws for reader monads to prove correctness properties about a modular compiler. In contrast to our work, their compiler correctness proofs are pen-and-paper and thus more informal than our proofs. Since they are not restricted by a termination checker or the use of positive types only, they exploit features like general recursion in their definitions. Oliveira et al. [2010] have also used algebraic laws for the state monad, in combination with parametricity, for modular proofs of non-interference of aspect-oriented advice. Gibbons and Hinze [2011] discuss several other algebraic laws for various types of monads. However, as far as we know, we are the first to provide an extensive mechanized library for monads and algebraic laws in Coq.

4.6.4 Mechanization of Monad Transformers

Huffman [2012] illustrates an approach for mechanizing type constructor classes in Isabelle/HOL with monad transformers. He considers transformer variants of the resumption, error and writer monads, but features only the generic functor, monad and transformer laws. The work tackles many issues that are not relevant for our Coq setting, such as lack of parametric polymorphism and explicit modeling of laziness.

4.7 Scientific Output

This chapter is based on the contents of the article

> Delaware, B., Keuchel, S., Schrijvers, T., and Oliveira, B. C. d. S. (2013). Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 319-330. ACM.

The project also drew ideas from previous unpublished work of mine (see [Keuchel and Schrijvers, 2012]):

> Keuchel, S. and Schrijvers, T. (2012). Modular Monadic Reasoning, a (Co-)Routine. Presented at *the 24th Symposium on Implementation and Application of Functional Languages*, IFL '12.

In particular, the modeling of explicit continuations in the monadic typing rules of the 3MT framework is reminiscent of a free monad or a coroutine/resumption monad which is the basis for this previous work. The coroutine based approach used an inductive datatype to encode computations which turned out to be cumbersome with respect to equality and algebraic reasoning. 3MT uses monadic values directly instead. Nevertheless, the fundamental setup of the use of explicit continuations in the monadic typing rules is directly influenced by my prior work. However, Benjamin Delaware took the lead in this project and has contributed the major part of the development and refinement of the monadic typing rules.

The reusable bind lemma was developed by me during the work on the case study and was used to significantly cut down the number of feature interaction lemmas.

Concerning the technical development in Coq, the 3MT monad library – including most of the algebraic laws – was written before I joined the project. My main contribution lies in the development of feature theorems, language compositions and the presentation of the case study in the publication.

Part II

Genericity

Chapter 5

Background

Names are commonly used in programming languages to refer to defined entities such as constants, functions, function parameters, classes, methods, record labels, program points, type and data constructors. *Variable binding* is a special case, in which names not only serve as references, but also as placeholder that can be *substituted* or instantiated with members of some domain. For instance, function parameters might be instantiated to values during evaluation, or to arbitrary expressions when the function is being inlined during compilation. On the other hand, we do normally not consider a Java class name to be substitutable, for example with the definition of the class.

Syntactic operations like calculating the set of free variables, renaming bound variables, or substituting variables can be defined for every language with variable binding. Moreover, their implementation is highly repetitive: it follows a pattern that only depends on the *scoping rules* of a particular language. Furthermore, proof of properties about these operations are equally repetitive.

This raises the question of whether we can achieve reuse in the implementation of such *boilerplate functions* and *boilerplate lemmas* related to variable binding. In this part of the thesis, we investigate a *generic approach* to solving this problem: generically defining the boilerplate and instantiating it to a particular language when needed.

This chapter covers background information on formalising languages with variable binding and illustrates the boilerplate that arises when mechanising meta-theory. Furthermore, it outlines and motivates our specific approach and

defines necessary terminology. Our running example is $F_{\exists,\times}$, i.e. System F with universal and existential quantification, and products. In the following sections, we elaborate on the development and point out which definitions and proofs can be considered *variable binding boilerplate* and which are *essential*. The kind of boilerplate that arises, is coarsely determined by the semantics, the meta-theoretic property that is being proved and the approach to proving it. We use the syntactic approach of Wright and Felleisen [1994] to prove type safety via progress and preservation of small-step operational semantics and specifically focus on the boilerplate of such proofs.

For the illustration in this chapter, we proceed in three steps. First, we present a textbook-like definition of $F_{\exists,\times}$. This definition is not completely formal, and hence cannot be used directly in mechanisations. It is, however, formal enough to be accepted in scientific publications. We call it *semi-formal*. Using this semi-formal definition we discuss arising boilerplate independent of later choices, e.g. syntax representations. Second, we formalise the previous semi-formal definition by bringing it into a shape that is suitable for mechanisation. Third, we discuss the mechanisation itself and conclude with an overview of the remainder of this part.

5.1 Semi-formal Development

This section presents the semi-formal development of the language $F_{\exists,\times}$, with an emphasis on variable binding related concerns. Section 5.1.1 presents the syntax of $F_{\exists,\times}$ and elaborates on needed variable binding boilerplate. Subsequently, Section 5.1.2 presents the typing and evaluation relations and illustrates the boilerplate lemmas they determine. Finally, Section 5.1.3 shows where the boilerplate is used in the type safety proof of $F_{\exists,\times}$.

5.1.1 Syntax

Like in Section 1.1.1, we define the syntax of $F_{\exists,\times}$ using an EBNF grammar. On top of this grammar, we define a substitution operation. We particularly pay attention to another concern related to variable binding, namely scoping rules, which we define using a well-scopedness relation. This is more formal and explicit than what is commonly found in textbooks.

Grammar Figure 5.1 shows the first part of the language specification: the definition of the syntax of $F_{\exists,\times}$, in a textbook-like manner.

α, β ::=	type variable	e ::=	term
τ, σ ::=	type	$\mid x$	term variable
$\mid \alpha$	type variable	$\mid \lambda x : \tau.e$	term abstraction
$\mid \sigma \to \tau$	function type	$\mid e_1\ e_2$	term application
$\mid \forall\alpha.\tau$	universal type	$\mid \Lambda\alpha.e$	type abstraction
$\mid \exists\alpha.\tau$	existential type	$\mid e!\tau$	type application
$\mid \sigma \times \tau$	product type	$\mid \{\sigma, e\}$ as τ	packing
x, y ::=	term variable	$\mid$ let $\{\alpha, x\} = e_1$ in e_2	unpacking
p ::=	pattern	$\mid e_1, e_2$	pair
$\mid x$	variable pattern	$\mid$ case e_1 of $p \to e_2$	pattern binding
$\mid p_1, p_2$	pair pattern	v ::=	value
Γ, Δ ::=	type context	$\mid \lambda x : \tau.e$	term abstraction
$\mid \epsilon$	empty context	$\mid \Lambda\alpha.e$	type abstraction
$\mid \Gamma, \alpha$	type binding	$\mid \{\sigma, v\}$ as τ	existential value
$\mid \Gamma, x : \tau$	term binding	$\mid v_1, v_2$	pair value

Figure 5.1: $F_{\exists,\times}$ syntax

The three main syntactic sorts of $F_{\exists,\times}$ are types, terms and patterns, and there are auxiliary sorts for values, variables and typing contexts. Patterns describe *pattern matching* for product types only and can be arbitrarily nested. A pattern can therefore bind an arbitrary number of variables at once. For simplicity, we keep matching on existentials separate from products. One level of existentials can be packed via ($\{\tau, e\}$ as σ) and unpacked via (let $\{\alpha, x\} = e_1$ in e_2).

In this grammar, the scoping rules are left implicit as is common practice. The intended rules are that in a universal ($\forall\alpha.\tau$) and existential quantification ($\exists\alpha.\tau$) the type variable α scopes over the body τ, in a type abstraction ($\Lambda\alpha.e$) and term abstraction ($\lambda x : \tau.e$) the variable α respectively x scopes over the body e. In a pattern binding (case e_1 of $p \to e_2$) the variables bound by the pattern p scope over e_2 but not e_1, and in the unpacking of an existential (let $\{\alpha, x\} = e_1$ in e_2) the variables α and x scope over e_2.

Term and type variables appear in two different modes in the productions for terms and types. First, in the variable production of each sort. We call this a *variable use* or *variable reference*. All other occurrences of term and type variable are *variable bindings*.

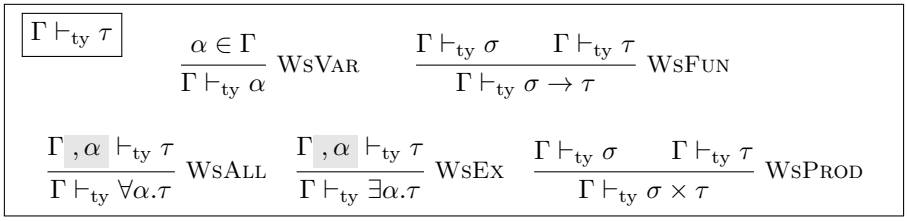


Figure 5.2: Well-scoping of types

Scoping We defined the scoping rules using prose above. We now give them a formal treatment by defining *well-scoping relations* that encode the scoping rules.

The well-scopedness relation for types $\Gamma \vdash_{\text{ty}} \tau$ is defined in Figure 5.2. This relation takes a typing context Γ as an index to represent the set of variables that are in scope and an index τ for types. It denotes that all type variables in τ are bound either in τ itself or appear in Γ. The definition is completely syntax-directed. The interesting rules are the ones for variables and for the quantifiers. In the variable case, rule WsVar checks that a type variable indeed appears in the typing context Γ. In the rules WsAll for universal and WsEx for existential quantification, the bound variable is added to the typing context in the premise for the bodies. Similar relations can be defined for terms, patterns and typing contexts as well.

The definition of well-scoping relations follows a standard recipe and usually its definition is left out in pen-and-paper specifications. In mechanisations, however, such a relation usually needs to be defined (unless it is not used in the meta-theory) by the human prover. Therefore, this relation is an example of syntax-related boilerplate that we want to derive generically. The structure of the relation only depends on the syntax of types and their scoping rules. But since the scoping rules are not reflected in the EBNF syntax of Figure 5.1 we need to add other pieces of *essential information* to the specification from which we can derive boilerplate.

One option is to make the scoping relations like $\Gamma \vdash_{\text{ty}} \tau$ part of the specification and derive other boilerplate from it, but this relation repeats a lot of information that is already given in the EBNF grammar. If possible, we want to avoid that repetition in our specifications. The only new detail that the well-scopedness relation adds explicitly, is that the type variables in the quantifications scope over the bodies, which is highlighted in gray in Figure 5.2.

$\boxed{\mathrm{fv}(\tau)}$

$$
\begin{array}{lcl}
\mathrm{fv}(\alpha) & = & \{\alpha\} \\
\mathrm{fv}(\tau_1 \to \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\
\mathrm{fv}(\forall\beta.\tau) & = & \mathrm{fv}(\tau) \setminus \{\beta\} \\
\mathrm{fv}(\exists\beta.\tau) & = & \mathrm{fv}(\tau) \setminus \{\beta\} \\
\mathrm{fv}(\tau_1 \times \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2)
\end{array}
$$

$\boxed{\mathrm{bnd}(p)}$

$$
\begin{array}{lcl}
\mathrm{bnd}(x) & = & \{x\} \\
\mathrm{bnd}(p_1, p_2) & = & \mathrm{bnd}(p_1) \cup \mathrm{bnd}(p_2)
\end{array}
$$

$\boxed{\mathrm{fv}(e)}$

$$
\begin{array}{lcl}
\mathrm{fv}(x) & = & \{x\} \\
\mathrm{fv}(\lambda x : \tau.e) & = & \mathrm{fv}(e) \setminus \{x\} \\
\mathrm{fv}(e_1\ e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\Lambda\alpha.e) & = & \mathrm{fv}(e) \setminus \{\alpha\} \\
\mathrm{fv}(e!\tau) & = & \mathrm{fv}(e) \cup \mathrm{fv}(\tau) \\
\mathrm{fv}(\{\sigma, e\} \text{ as } \tau) & = & \mathrm{fv}(\sigma) \cup \mathrm{fv}(e) \cup \mathrm{fv}(\tau) \\
\mathrm{fv}(\text{let } \{\alpha, x\} = e_1 \text{ in } e_2) & = & \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{\alpha, x\}) \\
\mathrm{fv}(e_1, e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\text{case } e_1 \text{ of } p \to e_2) & = & \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \mathrm{bnd}(p))
\end{array}
$$

Figure 5.3: Free variables

These issues ask to develop a new formal and concise way to specify scoping rules. We come back to this in Chapter 6 which presents our solution to the problem: we develop a language of (abstract) syntax specifications that include *binding specifications* for scoping.

Free Variables Figure 5.3 shows the definition of the calculations of free variables of types and terms, i.e. reference occurrences of variables that are not bound in the type or term itself. Free variables are used for the definition of capture-avoiding substitution below.

The implementation is a recursive traversal that accumulates free variables from the variable case leaves upward and removes variables when they are found to be bound. For sorts like patterns that represent binders we define an auxiliary function $\mathrm{bnd}(\cdot)$ that calculates the set of bound variables. The definition of the free variable function follows a standard recipe, which only depends on the grammar and the scoping rules of the syntactic sorts, and is

$\boxed{[\alpha \mapsto \sigma]\ \tau}$

$$
\begin{array}{lcll}
[\alpha \mapsto \sigma]\ \alpha & = & \sigma & \\
[\alpha \mapsto \sigma]\ \beta & = & \beta & (\alpha \neq \beta) \\
[\alpha \mapsto \sigma]\ (\tau_1 \to \tau_2) & = & ([\alpha \mapsto \sigma]\ \tau_1) \to ([\alpha \mapsto \sigma]\ \tau_2) & \\
[\alpha \mapsto \sigma]\ (\forall \beta.\tau) & = & \forall \beta.[\alpha \mapsto \sigma]\ \tau & (\alpha \neq \beta \wedge \beta \notin \mathrm{fv}(\sigma)) \\
[\alpha \mapsto \sigma]\ (\exists \beta.\tau) & = & \exists \beta.[\alpha \mapsto \sigma]\ \tau & (\alpha \neq \beta \wedge \beta \notin \mathrm{fv}(\sigma)) \\
[\alpha \mapsto \sigma]\ (\tau_1 \times \tau_2) & = & ([\alpha \mapsto \sigma]\ \tau_1) \times ([\alpha \mapsto \sigma]\ \tau_2) &
\end{array}
$$

Figure 5.4: Type in type substitutions

therefore boilerplate.

Substitution The typing and evaluation relations of $F_{\exists,\times}$ use type and term variable substitution which we define now. We need to define 3 substitution operators

$$[\alpha \mapsto \sigma]\ \tau \quad [\alpha \mapsto \sigma]\ e \quad [x \mapsto e_1]\ e_2$$

that correspond to substituting type variables in types and terms, and term variables in terms.

A correct definition of substitution is subtle when it comes to specific names of variables. A mere textual replacement is not sufficient. The following two examples illustrate situations where we expect a different result than a textual replacement:

$$
\begin{array}{lcl}
[\alpha \mapsto \sigma]\ (\Lambda\alpha.\alpha) & \neq & \Lambda\alpha.\sigma \\
[\alpha \mapsto (\sigma \to \beta)]\ (\Lambda\beta.\alpha) & \neq & \Lambda\beta.(\sigma \to \beta)
\end{array}
$$

In the first case, the type variable α is bound in a universal quantification in the type $(\Lambda\alpha.\alpha)$ we operate on. It should not have been substituted. The substitution should only substitute free variables. In the second case, we substitute the free variable α but another issue arises. The variable β that appears free in the substitute $(\sigma \to \beta)$ wrongly points to the β binder after replacement. This is commonly called a *variable capture*.

Figure 5.4 contains a definition of a (capture-avoiding) substitution operator that uses side-conditions to rule out the two problematic cases above. However, it rules out certain inputs and therefore makes the operations partial. This is widely accepted in semi-formal pen-and-paper proofs, but a stumbling block for mechanisation.

The partiality can be addressed by taking into account the intuition that the names of bound variables do not matter. For example, for our intended

semantics the terms $\lambda(x : \tau).x$ and $\lambda(y : \tau).y$ are essentially equivalent, i.e. we consider terms that are *equal up to consistent renaming of bound variables*. We can apply this in the definition of the substitution operators to replace bound variables with *fresh* ones, i.e. variables that are not used elsewhere. In other words, we perform α-conversion during substitution.

In pen-and-paper proofs, keeping track of α-conversions is onerous, detrimental to presentation, and utterly boring. Hence it is often assumed, that that at any time, all bound variables are distinct from free variables and implicitly renamed as needed. This is also called the *Barendregt variable convention* [Barendregt, 1984].

5.1.2 Semantics

The next step in the formalization is to develop the typical semantic relations for the language of study. In the case of $F_{\exists,\times}$, these comprise a typing relation for terms, a typing relation for patterns, and a small-step call-by-value operational semantics.

Typing Figure 5.5 contains the rules for the term and pattern typing relations. The variable rule TVar of the term typing looks up a term variable x with its associated type τ in the typing context Γ and rule TAbs deals with abstractions over terms in terms which adds the binding $(y : \sigma)$ to the typing context for the premise of the body e. The rules TTApp for type-application and TPack for packing existential types use a type-substitution operation $[\alpha \mapsto \sigma]\tau$ that substitutes σ for α in τ. TTApp performs the substitution in the conclusion while TPack does so in the premise. The remaining two rules, TPair and TCase, of the term typing relation deal with products. In a case expression the pattern p needs to have the same type σ as the scrutinee e_1 and the variables Δ bound by p are brought into scope in the body e_2. This environment Δ is the output of the pattern typing relation $\Gamma \vdash_{\mathrm{p}} p : \tau; \Delta$, which contains the typing information for all variables bound by p. This information is concatenated in the rule PPair for pair patterns.

Evaluation The operational semantics is defined with 4 reduction rules shown in Figure 5.6. We omitted further congruence rules that determine the evaluation order. The reduction of the case construct uses an auxiliary pattern-matching relation Match $v\ p\ e_1\ e_2$ which denotes that when matching the value v against the pattern p and applying the resulting variable substitution to e_1 we get e_2 as a result. All of the reduction rules directly or indirectly use substitutions.

$\boxed{\Gamma \vdash_{\text{tm}} e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{tm}} x : \tau}\ \textsc{TVar}$$

$$\frac{\Gamma, y : \sigma \vdash_{\text{tm}} e : \tau}{\Gamma \vdash_{\text{tm}} (\lambda y : \sigma.e) : (\sigma \to \tau)}\ \textsc{TAbs} \qquad \frac{\Gamma, \alpha \vdash_{\text{tm}} e : \tau}{\Gamma \vdash_{\text{tm}} (\Lambda\alpha.e) : (\forall\alpha.\tau)}\ \textsc{TTAbs}$$

$$\frac{\Gamma \vdash_{\text{tm}} e_1 : \sigma \to \tau \quad \Gamma \vdash_{\text{tm}} e_2 : \sigma}{\Gamma \vdash_{\text{tm}} (e_1\ e_2) : \tau}\ \textsc{TApp} \qquad \frac{\Gamma \vdash_{\text{tm}} e : \forall\alpha.\tau}{\Gamma \vdash_{\text{tm}} (e!\sigma) : ([\alpha \mapsto \sigma]\tau)}\ \textsc{TTApp}$$

$$\frac{\Gamma \vdash_{\text{tm}} e : ([\alpha \mapsto \sigma]\tau)}{\Gamma \vdash_{\text{tm}} (\{\sigma, e\} \text{ as } \exists\alpha.\tau) : (\exists\alpha.\tau)}\ \textsc{TPack}$$

$$\frac{\Gamma \vdash_{\text{tm}} e_1 : \tau_1 \qquad \Gamma \vdash_{\text{tm}} e_2 : \tau_2}{\Gamma \vdash_{\text{tm}} (e_1, e_2) : (\tau_1 \times \tau_2)}\ \textsc{TPair}$$

$$\frac{\Gamma \vdash_{\text{tm}} e_1 : \exists\alpha.\tau \qquad \Gamma, \alpha, x : \tau \vdash_{\text{tm}} e_2 : \sigma \qquad \alpha \notin \text{fv}(\sigma)}{\Gamma \vdash_{\text{tm}} (\text{let } \{\alpha, x\} = e_1 \text{ in } e_2) : \sigma}\ \textsc{TUnpack}$$

$$\frac{\Gamma \vdash_{\text{tm}} e_1 : \sigma \qquad \Gamma \vdash_{\text{p}} p : \sigma; \Delta \qquad \Gamma, \Delta \vdash_{\text{tm}} e_2 : \tau}{\Gamma \vdash_{\text{tm}} (\text{case } e_1 \text{ of } p \to e_2) : \tau}\ \textsc{TCase}$$

$\boxed{\Gamma \vdash_{\text{p}} p : \tau; \Delta}$

$$\frac{}{\Gamma \vdash_{\text{p}} x : \tau; (\epsilon, x : \tau)}\ \textsc{PVar}$$

$$\frac{\Gamma \vdash_{\text{p}} p_1 : \tau_1; \Delta_1 \qquad \Gamma, \Delta_1 \vdash_{\text{p}} p_2 : \tau_2; \Delta_2}{\Gamma \vdash_{\text{p}} (p_1, p_2) : (\tau_1 \times \tau_2); (\Delta_1, \Delta_2)}\ \textsc{PPair}$$

Figure 5.5: $F_{\exists,\times}$ typing rules

5.1.3 Meta-Theory

We conclude the semi-formal development by discussing the meta-theoretic proof of type safety for $F_{\exists,\times}$. We do not go into the details of the proof, but focus instead on the variable binding related proof steps.

$\boxed{e \longrightarrow e}$

$$\frac{}{(\lambda x.e)\ v \longrightarrow [x \mapsto v]e} \qquad \frac{}{(\Lambda\alpha.e)!\tau \longrightarrow [\alpha \mapsto \tau]e}$$

$$\frac{}{\text{let } \{\alpha, x\} = \{\sigma, v\} \text{ as } \tau \text{ in } e \longrightarrow [\alpha \mapsto \sigma][x \mapsto v]e}$$

$$\frac{\text{Match } v\ p\ e_1\ e_2}{(\text{case } v \text{ of } p \to e_1) \longrightarrow e_2}$$

$\boxed{\text{Match } v\ p\ e_1\ e_2}$

$$\frac{}{\text{Match } v\ x\ e\ ([x \mapsto v]e)}$$

$$\frac{\text{Match } v_1\ p_1\ e_1\ e_2 \qquad \text{Match } v_2\ p_2\ e_2\ e_3}{\text{Match } (v_1, v_2)\ (p_1, p_2)\ e_1\ e_3}$$

Figure 5.6: $F_{\exists,\times}$ evaluation - selected rules

Substitution The interesting steps in the type preservation proof are the preservations under the 4 reduction rules of the operational semantics. These essentially boil down down to two substitution lemmas:

$$\frac{\Gamma \vdash_{\text{tm}} e_1 : \sigma \qquad \Gamma, x : \sigma, \Delta \vdash_{\text{tm}} e_2 : \tau}{\Gamma, \Delta \vdash_{\text{tm}} [x \mapsto e_1]e_2 : \tau}\ \textsc{SubstTmTm}$$

$$\frac{\Gamma, \beta, \Delta \vdash_{\text{tm}} e : \tau}{\Gamma, [\beta \mapsto \sigma]\Delta \vdash_{\text{tm}} [\beta \mapsto \sigma]e : [\beta \mapsto \sigma]\tau}\ \textsc{SubstTyTm}$$

For the proofs by induction of these lemmas to go through, we need to prove them for all suffixes Δ, but only use the special case where $\Delta = \epsilon$ in the preservation proof. For the inductive step for rule TTApp of the second substitution lemma we have to prove the following

$$\frac{\Gamma' = \Gamma, [\beta \mapsto \sigma]\Delta \qquad \Gamma' \vdash_{\text{tm}} [\beta \mapsto \sigma]e : \forall\alpha.[\beta \mapsto \sigma]\tau}{\Gamma' \vdash_{\text{tm}} ([\beta \mapsto \sigma]e)!([\beta \mapsto \sigma]\sigma') : \ [\beta \mapsto \sigma][\alpha \mapsto \sigma']\tau}$$

As the term in the conclusion remains a type application, we want to apply rule TTApp again. However, the type in the conclusion does not have the

appropriate form. We first need to commute the two substitutions with one of the common interaction lemmas

$$[\beta \mapsto \sigma][\alpha \mapsto \sigma'] = [\alpha \mapsto [\beta \mapsto \sigma]\sigma'][\beta \mapsto \sigma] \qquad (\alpha \neq \beta) \tag{5.1}$$

Progress The proof of progress proceeds similarly to that in Section 1.2. The main difference, due to the presence of pattern matching, is that we need to prove an auxiliary lemma stating that well-typed pattern matching Match $v\ p\ e_1\ e_2$ is always defined:

$$\frac{\epsilon \vdash_{\text{tm}} v : \sigma \qquad \epsilon \vdash_{\text{p}} p : \sigma; \Delta \qquad \Delta \vdash_{\text{tm}} e_1 : \tau \qquad \epsilon \vdash_{\text{tm}} e_2 : \tau}{\text{Match } v\ p\ e_1\ e_2}$$

On the whole, the proof of progress does not involve a lot of semantic variable binding boilerplate.

Preservation The proof of preservation proceeds by induction on the typing derivation $\Gamma \vdash_{\text{tm}} e : \sigma$ and inversion of the evaluation relation $e \longrightarrow e'$. There are two kinds of cases. Firstly, for evaluation steps that are congruence rules, the proof follows immediately by applying the same rule again. Secondly, for the reduction rules, which all involve substitution, the proof involves boilerplate substitution lemmas. Consider the case of reducing a term abstraction

$$(\lambda(x : \sigma).e_1)\ e_2 \longrightarrow [x \mapsto e_2]\ e_1.$$

The proof obligation is

$$\frac{\Gamma \vdash_{\text{tm}} \lambda(x : \sigma).e_1 : \sigma \to \tau \qquad \Gamma \vdash_{\text{tm}} e_2 : \sigma}{\Gamma \vdash_{\text{tm}} [x \mapsto e_2]e_1 : \tau}$$

After inverting the first premise $\Gamma \vdash_{\text{tm}} \lambda(x : \sigma).e_1 : \tau$ to get the typing derivation of e_1 we can apply the boilerplate lemma for well-typed substitutions SubstTmTm, with $\Delta = \epsilon$, to finish the proof of this case.

Similarly, the cases of universal and existential quantification follow directly from the typing substitution lemmas. The case of a pattern match additionally requires an induction over the matching relation Match $v\ p\ e_1\ e_2$. The boilerplate lemma SubstTmTm is used in the pattern variable case.

5.2 Formalization and Mechanisation

The next step is to rework the textbook-like specification from Section 5.1 into a formal one, which can be mechanised in a proof assistant. In the following,

$$
\begin{array}{lllll}
T & ::= & n & | & \forall \bullet . T \\
 & | & T_1 \to T_2 & | & \exists \bullet . T \\
 & | & T_1 \times T_2 & | &
\end{array}
\qquad
\begin{array}{lll}
E & ::= & \epsilon \\
 & | & \Gamma, \bullet \\
 & | & \Gamma, \bullet : T
\end{array}
$$

$$
\begin{array}{lllllll}
q & ::= & \bullet & t & ::= & n & | & \Lambda \bullet . t & | & t_1, t_2 \\
 & | & q_1, q_2 & & | & \lambda \bullet : T . t & | & t ! T & | & \text{case } t_1 \text{ of } q \to t_2 \\
 & & & & | & t_1\ t_2 & | & \{T_1, t\} \text{ as } T_2 & | & \text{let } \{\bullet, \bullet\} = t_1 \text{ in } t_2
\end{array}
$$

Figure 5.7: $F_{\exists,\times}$ de Bruijn representation

we will replace the syntax representation and discuss changes to the semantics definitions. Finally, we discuss a mechanisation itself, and give a breakdown of the different parts of the mechanisation to quantify the overall effort and particularly the burden of variable binding boilerplate.

5.2.1 Syntax Representation

The first step in the mechanisation is to choose how to concretely represent variables. Traditionally, one would represent variables using identifiers, but as indicated in the previous section, capture-avoidance requires α-conversion and keeping track of conversions is tedious. In the semi-formal development we allowed ourselves the luxury of using the Barendregt variable convention. However, the Barendregt convention comes with it own subtleties that can even result in faulty reasoning [Urban et al., 2007; Pitts, 2003]. Hence, it undermines our goal for rigorous, gapless, and machine-checked proofs.

Instead, the issue of α-conversion is handled by moving to a different representations. Possible candidates for our formalization are de Bruijn index-based [de Bruijn, 1972], locally nameless [Aydemir et al., 2008] or locally named [Sato and Pollack, 2010; Pollack et al., 2012] representations that *canonically represent α-equivalent terms* and thus eliminate α-conversion altogether. An alternative is *nominal abstract syntax*[Pitts, 2003] that represents terms using α-equivalance classes for which induction and recursion principles can be defined. Higher-order abstract syntax (HOAS) [Pfenning and Elliott, 1988] is a radically different approach that is using binders of the meta-language to represent binders of an object language.

Our goal is not to develop a new approach to representing syntax with variable binding nor to compare existing ones, but rather to scale the generic treatment of a single approach. For this purpose, we choose de Bruijn representations [de Bruijn, 1972], motivated by two main reasons. First, reasoning with

de Bruijn representations is well-understood and, in particular, the representation of pattern binding and scoping rules is also well-understood [de Bruijn, 1991; Keuchel and Jeuring, 2012]. Second, the functions related to variable binding, the statements of properties of these functions and their proofs have highly regular structures with respect to the abstract syntax and the scoping rules of the language. This helps us in treating boilerplate generically and automating proofs.

Figure 5.7 shows a term grammar for a de Bruijn representation of $F_{\exists,\times}$. A property of this representation is that binding occurrences of variables still mark the binding but do not explictly name the bound variable anymore. For this reason, we replace the variable names in binders uniformly with a bullet $\bullet$ in the new grammar. Variables do not refer to their binding site by name but by using positional information: A variable is represented by a natural number n that denotes that the variable is bound by the nth enclosing binder starting from 0.

Here are several examples of terms in both the named and the de Bruijn representation:

$$\begin{array}{lcl}
\lambda(x:\tau).x & \Rightarrow & (\lambda(\bullet:T).0) \\
\lambda(x:\tau).\lambda(y:\sigma).x & \Rightarrow & (\lambda(\bullet:T).\lambda(\bullet:S).1) \\
\lambda(x:\tau_1).\lambda(y:\tau_1\to\tau_2).(\lambda(z:\tau_1).y\ x)\ x & \Rightarrow & \\
\qquad \lambda(\bullet:T_1).\lambda(\bullet:T_1\to T_2).(\lambda(\bullet:T_1).2\ 0)\ 1 & &
\end{array}$$

In the first example, $\lambda(x:\tau).x$, the variable x refers to the immediately enclosing binding and can therefore be represented with the index 0. Next, in $\lambda(x:\tau).\lambda(y:\sigma).x$, we need to skip the y binding and therefore represent the occurrence of x with 1. In the third example, the variable x is once represented using the index 1 and once using the index 2. This shows that the indices of a variable are not constant but depend on the context the variable appears in.

Finally, we use different namespaces for term and type variables and treat indices for variables from distinct namespaces independently, as illustrated by the following examples:

$$\begin{array}{lcl}
\lambda(x:\tau).\Lambda\alpha.x!\alpha & \Rightarrow & \lambda(\bullet:\tau).\Lambda\bullet.0!0 \\
\Lambda\alpha.\lambda(x:\tau).x!\alpha & \Rightarrow & \Lambda\bullet.\lambda(\bullet:\tau).0!0 \\
\Lambda\alpha.\Lambda\beta.\lambda(x:\alpha).\lambda(y:\beta).x & \Rightarrow & \Lambda\bullet.\Lambda\bullet.\lambda(\bullet:1).\lambda(\bullet:0).1
\end{array}$$

As a consequence, when resolving a term variable index we do not take type variable binders into account and vice versa.

5.2.2 Well-scopedness

In the semi-formal specification we have defined a well-scopedness relation solely for the purpose to make the scoping rules explicit. Of course, we only ever want to consider well-scoped terms which is implicitly assumed in the semi-formal development. In a proper formalization and mechanisation, however, this introduces obligation to actually prove terms to be well-scoped. For instance, we need to prove that that all syntactic operations, like substitution, preserve well-scopedness.

The well-scopedness of de Bruijn terms is a syntactic concern. It is common practice to define well-scopedness with respect to a *type* context like we did in Section 5.1.1 : a term is well-scoped iff all its free variables are bound in the context. The context is extended when going under binders. For example, when going under the binder of a type-annotated lambda abstraction the conventional rule is:

$$\frac{\Gamma, x : \tau \vdash_{\mathrm{tm}} e}{\Gamma \vdash_{\mathrm{tm}} \lambda x : \tau . e}$$

The rule follows the intention that the term variable should be of the given type. In this style, well-scopedness comprises a lightweight type system. However, in general it is impossible to come up with the intended typing or, more generally, establish what the associated data in the extended context should be. Furthermore, we allow the user to define different contexts with potentially incompatible associated data. To avoid this issue, we define well-scopedness by using *domains* of contexts instead. In fact, this is all we need to establish well-scopedness.

In a de Bruijn approach the domain is traditionally represented by a natural number that denotes the number of bound variables. Instead, we use *heterogeneous numbers* h – a refinement of natural numbers – defined in Figure 5.8 to deal with heterogeneous contexts: each successor is tagged with a namespace to keep track of the number and order of variables of different namespaces. This also allows us to model languages with heterogeneous binders, i.e. that bind variables of different namespaces at the same time, for which reordering the bindings is undesirable. In the following, we abbreviate units such as $1_{\mathrm{ty}} := \mathrm{S}_{\mathrm{ty}}\ 0$, use the obvious extension of addition from natural numbers to heterogeneous numbers and implicitly use its associativity property. In contrast to naturals, addition is not commutative. We mirror the convention of extending contexts to the right at the level of h and will always add new variables on the right-hand side.

Figure 5.8 also defines the calculation dom of domains of typing contexts, a well-scopedness predicate $h \vdash_{\mathrm{tm}} n$ for term indices, which corresponds to

$$h \quad ::= \quad 0 \mid S_{\text{ty}}\ h \mid S_{\text{tm}}\ h$$

$\boxed{\text{dom} : E \to h}$

$$\begin{aligned} \text{dom}\ \epsilon &= 0 \\ \text{dom}\ (E, \bullet) &= \text{dom}\ E + 1_{\text{ty}} \\ \text{dom}\ (E, \bullet : T) &= \text{dom}\ E + 1_{\text{tm}} \end{aligned}$$

$\boxed{h \vdash^{\text{var}}_{\text{tm}} n}$

$$\frac{}{S_{\text{tm}}\ h \vdash^{\text{var}}_{\text{tm}} 0}\ \textsc{WsnTmZero}$$

$$\frac{h \vdash^{\text{var}}_{\text{tm}} n}{S_{\text{tm}}\ h \vdash^{\text{var}}_{\text{tm}} S\ n}\ \textsc{WsnTmTm}$$

$$\frac{h \vdash^{\text{var}}_{\text{tm}} n}{S_{\text{ty}}\ h \vdash^{\text{var}}_{\text{tm}} n}\ \textsc{WsnTmTy}$$

$\boxed{h \vdash_{\text{tm}} t}$

$$\frac{h \vdash^{\text{var}}_{\text{tm}} n}{h \vdash_{\text{tm}} n}\ \textsc{WsVar}$$

$$\frac{h \vdash_{\text{tm}}\ t_1 \qquad h + 1_{\text{ty}} + 1_{\text{tm}} \vdash_{\text{tm}}\ t_2}{h \vdash_{\text{tm}} (\text{let}\ \{\bullet, \bullet\} = t_1\ \text{in}\ t_2)}\ \textsc{WsUnpack}$$

$\boxed{h \vdash E}$

$$\frac{h \vdash E \qquad h + \text{dom}\ E \vdash T}{h \vdash E, \bullet : T}\ \textsc{WseTm}$$

Figure 5.8: Well-Scopedness of Terms (selected rules)

$n < h$ when only term successors are counted, and a selection of rules for well-scopedness of terms $h \vdash_{\text{tm}} t$ and well-scopedness of typing environments $h \vdash E$.

5.2.3 Substitutions

The operational semantics and typing relations of $F_{\exists,\times}$ require boilerplate definitions for the de Bruijn representation: substitution of type variables in types, terms and type contexts, and of term variables in terms. We also need to define four auxiliary boilerplate *shifting* functions that adapt indices of free variables when going under binders, or, put differently, when inserting new

$\boxed{\text{shift}_{\text{tm}} : h \to n \to n}$

$$
\begin{array}{llll}
\text{shift}_{\text{tm}} & 0 & n & = & S\ n \\
\text{shift}_{\text{tm}} & (S_{\text{tm}}\ h) & 0 & = & 0 \\
\text{shift}_{\text{tm}} & (S_{\text{tm}}\ h) & (S\ n) & = & S\ (\text{shift}_{\text{tm}}\ h\ n) \\
\text{shift}_{\text{tm}} & (S_{\text{ty}}\ h) & n & = & \text{shift}_{\text{tm}}\ h\ n
\end{array}
$$

$\boxed{\text{shift}_{\text{tm}} : h \to t \to t}$

$$
\begin{array}{llll}
\text{shift}_{\text{tm}} & h & (\Lambda \bullet .t) & = & \Lambda \bullet .(\text{shift}_{\text{tm}}\ (S_{\text{ty}}\ h)\ t) \\
\text{shift}_{\text{tm}} & h & (t_1, t_2) & = & (\text{shift}_{\text{tm}}\ h\ t_1), (\text{shift}_{\text{tm}}\ h\ t_2) \\
\text{shift}_{\text{tm}} & h & (\lambda \bullet : T.t) & = & \lambda \bullet : T.(\text{shift}_{\text{tm}}\ (S_{\text{tm}}\ h)\ t) \\
\text{shift}_{\text{tm}} & h & (t!T) & = & (\text{shift}_{\text{tm}}\ h\ t)!T \\
\text{shift}_{\text{tm}} & h & (\text{case}\ t_1\ \text{of}\ q \to t_2) & = & \\
\multicolumn{4}{l}{\quad \text{case}\ (\text{shift}_{\text{tm}}\ h\ t_1)\ \text{of}\ q \to (\text{shift}_{\text{tm}}\ (h + \text{bnd}(q))\ t_2)} \\
\text{shift}_{\text{tm}} & h & (t_1\ t_2) & = & (\text{shift}_{\text{tm}}\ h\ t_1)\ (\text{shift}_{\text{tm}}\ h\ t_2) \\
\text{shift}_{\text{tm}} & h & (\{T_1, t\}\ \text{as}\ T_2) & = & \{T_1, \text{shift}_{\text{tm}}\ h\ t\}\ \text{as}\ T_2 \\
\text{shift}_{\text{tm}} & h & (\text{let}\ \{\bullet, \bullet\} = t_1\ \text{in}\ t_2) & = & \\
\multicolumn{4}{l}{\quad \text{let}\ \{\bullet, \bullet\} = (\text{shift}_{\text{tm}}\ h\ t_1)\ \text{in}\ (\text{shift}_{\text{tm}}\ (h + 1_{\text{ty}} + 1_{\text{tm}})\ t_2)}
\end{array}
$$

$\boxed{\text{shift}_{\text{ty}} : h \to E \to E}$

$$
\begin{array}{llll}
\text{shift}_{\text{ty}} & h & \epsilon & = & \epsilon \\
\text{shift}_{\text{ty}} & h & (E, \bullet) & = & (\text{shift}_{\text{ty}}\ h\ E), \bullet \\
\text{shift}_{\text{ty}} & h & (E, \bullet : T) & = & (\text{shift}_{\text{ty}}\ h\ E), \bullet : (\text{shift}_{\text{ty}}\ (h + \text{dom}\ E)\ T)
\end{array}
$$

Figure 5.9: Shifting functions

variables in the context.

Shifting *Shifting* is an operation that adapts indices in terms when the context is extended with new variables, e.g. when recursing under a binder during substitution. For example, when going under a term or a type abstraction we need to transport the substitutes along the following context changes:

$$\Gamma \vdash e \rightsquigarrow \Gamma, x \vdash e \quad \text{and} \quad \Gamma \vdash e \rightsquigarrow \Gamma, \alpha \vdash e.$$

To implement shiftings, we need to generalize them first, so that variables can be inserted in the middle of the context, i.e. operations that correspond to the context changes

$$\Gamma, \Delta \vdash e \rightsquigarrow \Gamma, x, \Delta \vdash e \quad \text{and} \quad \Gamma, \Delta \vdash e \rightsquigarrow \Gamma, \alpha, \Delta \vdash e.$$

Only indices for variables in Γ need to be adapted. For this purpose the shifting functions take a *cut-off* parameter that represents the domain of Δ. Only indices "above" the cut-off are adapted. We overload the name of type variable shifting and hence use the following four shift functions:

$$\boxed{\mathrm{shift}_{\mathrm{tm}} : h \to t \to t} \qquad \boxed{\mathrm{shift}_{\mathrm{ty}} : h \to E \to E}$$
$$\boxed{\mathrm{shift}_{\mathrm{ty}} : h \to t \to t} \qquad \boxed{\mathrm{shift}_{\mathrm{ty}} : h \to T \to T}$$

Figure 5.9 defines selected shiftings: shifting of term variables on terms and term indices, and of type variables in typing environments.

Instead of using the traditional arithmetical implementation

$$\textbf{if } n < c \textbf{ then } n \textbf{ else } n + 1$$

for the shifting of indices, we use an equivalent recursive definition that inserts the successor constructor *at the right place*. This follows the inductive structure of Δ which facilitates inductive proofs on Δ.

The shiftings can be iterated to get weakenings of multiple variables at once. In the essential meta-theoretic lemmas we will only use this form of weakening and will call it *lifting*, i.e. we can define functions

$$\mathit{lift} : t \to h \to t \quad \mathit{lift} : T \to h \to T \quad \mathit{lift} : E \to h \to E$$

that represent the weakenings

$$\frac{h \vdash t}{h, h^+ \vdash \mathit{lift}\ t\ h^+} \qquad \frac{h \vdash T}{h, h^+ \vdash \mathit{lift}\ T\ h^+} \qquad \frac{h \vdash E'}{h, h^+ \vdash \mathit{lift}\ E'\ h^+}$$

Substitution Next, we define substitution of a single variable x for a term e in some other term e' generically. In the literature, two commonly used variants can be found.

1. The first variant keeps the invariant that e and e' are in the same context and immediately weakens e when passing under a binder while traversing e' to keep this invariant. It corresponds to the substition lemma

$$\frac{\Gamma, \Delta \vdash e : \sigma \qquad \Gamma, x : \sigma, \Delta \vdash e' : \tau}{\Gamma, \Delta \vdash \{x \mapsto e\}\ e' : \tau}$$

2. The second variant keeps the invariant that e' is in a weaker context than e. It defers weakening of e until the variable positions are reached

to keep the invariant and performs shifting if the variable is substituted. It corresponds to the substitution lemma

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma, \Delta \vdash e' : \tau}{\Gamma, \Delta \vdash [x \mapsto e]\ e' : \tau}$$

Both variants were already present in de Bruijn's seminal paper [de Bruijn, 1972], but the first variant has enjoyed more widespread use. However, we will use the second variant because it has the following advantages:

1. It supports the more general case of languages with a dependent context:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma, \Delta \vdash e' : \tau}{\Gamma, [x \mapsto e]\ \Delta \vdash [x \mapsto e]\ e' : [x \mapsto e]\ \tau}$$

2. The parameter e is constant while recursing into e' and hence it can also be moved outside of inductions on the structure of e. Proofs become slightly simpler because we do not need to reason about any changes to s when going under binders.

We use substitution functions which keep the substitute in its original context and perform (multi-place) shifting when reaching the variable positions. This behaviour corresponds to the structure of the substitution lemmas SubstTmTm and SubstTyTm. Hence, the index to be substituted is represented by the domain of the suffix Δ to have enough information for the shifting.

$\mathrm{subst}_{\mathrm{tm}} : h \to t \to t \to t$	$\mathrm{subst}_{\mathrm{ty}} : h \to T \to T \to T$
$\mathrm{subst}_{\mathrm{ty}} : h \to T \to t \to t$	$\mathrm{subst}_{\mathrm{ty}} : h \to T \to E \to E$

5.2.4 Semantic Representation

The semantic typing relation from Figure 5.1 translates almost directly to a relation on the de Bruijn representation. One important aspect that is ignored in Figure 5.1 is to ensure that all rule components are well-scoped. This requires including additional well-scopedness premises in the rules. The rules that need an additional premise are shown in Figure 5.10 and the new premises have been highlighted.

We want to ensure that all appearing terms are well-scoped. For this it suffices to know that the objects represented by meta-variables are well-scoped. For instance, the meta-variables S, T and t in the TAbs rule. However, we

$\boxed{E \vdash_{\text{tm}} t : T}$

$$\frac{E \vdash_{\text{ty}} S \qquad E, \bullet : S \vdash_{\text{tm}} t : T}{E \vdash_{\text{tm}} (\lambda \bullet : S.t) : (S \to T)}\ \textsc{TAbs}$$

$$\frac{E \vdash_{\text{tm}} t : \forall \bullet .T \qquad E \vdash_{\text{ty}} S}{E \vdash_{\text{tm}} (e!S) : (\text{subst}_{\text{ty}}\ 0\ S\ T)}\ \textsc{TTApp}$$

$$\frac{E \vdash_{\text{tm}} t_1 : \exists \bullet .T \qquad E, \bullet, \bullet : T \vdash_{\text{tm}} e_2 : S \qquad E \vdash_{\text{ty}} S}{E \vdash_{\text{tm}} (\text{let } \{\alpha, x\} = e_1 \text{ in } e_2) : S}\ \textsc{TUnpack}$$

$\boxed{E \vdash_{\text{p}} q : T; D}$

$$\frac{E \vdash_{\text{ty}} T}{E \vdash_{\text{p}} \bullet : T; (\epsilon, \bullet : T)}\ \textsc{PVar}$$

Figure 5.10: $F_{\exists,\times}$ typing rules (de Bruijn, selected rules)

only included an explicit premise for S. The well-scopedness of T and t follows from two boilerplate lemmas:

$$\frac{0 \vdash E \qquad E \vdash_{\text{tm}} t : T}{E \vdash_{\text{tm}} t}\ \textsc{TypingScopeTm} \qquad \frac{\vdash E \qquad E \vdash_{\text{tm}} t : T}{E \vdash_{\text{ty}} T}\ \textsc{TypingScopeTy}$$

These lemmas express that well-typed terms are also well-scoped with a well-scoped type. More generally, we require that all semantic relations imply well-scoping of their indices. If the well-scoping of a meta-variable is not implied by a premise, an explicit well-scoping requirement needs to be added.

5.2.5 Meta-Theory

The essential meta-theoretic lemmas are only slightly affected by the changes to the syntax and semantics: the specific statements have to be adapted to the changed representation and some premises need additional premises, e.g. context well-formedness, but the structure of their proofs, i.e. the specific proof steps, remain the same. We therefore focus on the boilerplate lemmas, of which there are two kinds: syntactic related boilerplate and semantic related

boilerplate.

Syntactic Boilerplate

The principle syntactic operations that we use in the development are

- concatenate two context,
- calculate the domain of contexts,
- shifting in types, terms and contexts,
- and substitution in types, terms and contexts.

The syntactic boilerplate lemmas concern these syntactic operations. As already discussed we need to show that they preserve well-scopedness, but we also need to prove *interaction lemmas* between two or more of these operations. In Section 5.1.3 we briefly mentioned one instance of an interaction lemma: the commutation of two type-variable substitutions. Similarly, we need to prove commutation of e.g. shifting types and substituting terms, calculation of the domain of shifted contexts, commute substitution over a concatenation of contexts, etc. Moreover, there is a plethora of trivial lemmas, like the associativity of context concatenation, that are also interaction lemma. The interaction lemmas are all small but come in large numbers.

Semantic Boilerplate

The semantic boilerplate concerns the semantic relations. Only lemmas about the typing relation are needed. These are the two well-scoping lemmas we discussed in Section 5.2.4

$$\frac{\vdash E \qquad E \vdash_{\mathrm{tm}} t : T}{E \vdash_{\mathrm{tm}} t} \qquad \frac{\vdash E \qquad E \vdash_{\mathrm{tm}} t : T}{E \vdash_{\mathrm{ty}} T}$$

two shifting lemmas

$$\frac{E \vdash_{\mathrm{tm}} t : T}{E, \bullet : T' \vdash_{\mathrm{tm}} (\mathrm{shift}_{\mathrm{tm}}\ 0\ t) : T} \qquad \frac{E \vdash_{\mathrm{tm}} t : T}{E, \bullet \vdash_{\mathrm{tm}} (\mathrm{shift}_{\mathrm{ty}}\ 0\ t) : (\mathrm{shift}_{\mathrm{ty}}\ 0\ T)}$$

	Useful	Boilerplate
Specification	123 (13.3%)	164 (17.8%)
Syntax Theory	0 (0.0%)	365 (39.6%)
Semantics Theory	101 (11.0%)	187 (20.3%)
Total	224 (24.3%)	716 (77.7%)

Table 5.1: Lines of Coq code for the $F_{\exists,\times}$ meta-theory mechanisation.

and two substitution lemmas

$$\frac{0 \vdash E \qquad E \vdash_{\mathrm{tm}} t_1 : T_1 \qquad E, \bullet : T_1 \vdash_{\mathrm{tm}} t_2 : T_2}{E \vdash_{\mathrm{tm}} (\mathrm{subst}_{\mathrm{tm}}\ 0\ t_1\ t_2) : T_2}$$

$$\frac{0 \vdash E \qquad \mathrm{dom}\ E \vdash T_1 \qquad E, \bullet \vdash_{\mathrm{tm}} t_2 : T_2}{E \vdash_{\mathrm{tm}} (\mathrm{subst}_{\mathrm{ty}}\ 0\ T_1\ t_2) : (\mathrm{subst}_{\mathrm{ty}}\ 0\ T_1\ T_2)}$$

For the induction, the shifting and substitution lemmas need to be generalized to work with under arbitrary suffix Δ and require extensive use of the interaction lemmas.

5.2.6 Mechanisation

Table 5.1 summarizes the effort required to mechanise $F_{\exists,\times}$ in the Coq proof assistant in terms of the de Bruijn representation. It lists the lines of Coq code for different parts divided in binder-related *boilerplate* and other *useful* code. The *specification* row shows the code necessary to fully specify the syntax and semantics. The boilerplate that arises in this part are the shifting and substitution functions, context lookups and well-scopedness predicates that are necessary to define typing and operational semantics. The *syntax-related theory* consists of boilerplate lemmas like the commutation lemma (5.1) for type-substitutions. The useful *semantics-related theory* are canonical forms, typing inversion, progress and preservation lemmas. The boilerplate in this part are the well-scopedness, shifting and substitution lemmas for the typing relation of Section 5.2.

Summary Table 5.1 clearly shows that the boilerplate constitutes the major part of the effort. Similar boilerplate arises in the formalization of other languages where it constitutes a similar large part of the whole formalization.

5.3 Our Approach

As we illustrated in this chapter, the variable binding boilerplate puts a dolorous burden on formal mechanised meta-theory of languages. Fortunately, there is much regularity to the boilerplate: it follows the structure of the language's syntax, its scoping rules and the structure of expressions in rules of the semantic relations. This fact has already been exploited by many earlier works to derive *syntax-related* boilerplate functions and lemmas.

The aim of this thesis is to considerably extend the support for binder boilerplate in language mechanizations on two accounts. First, we go beyond simple single variable binders and tackle complex binding structures, like the nested pattern matches of $F_{\exists,\times}$, sequentially scoped binders, mutually recursive binders, heterogeneous binders, etc. Secondly, we cover a larger extent of the boilerplate than earlier works, specifically catering to contexts, context lookups and well-scopedness relations.

Our approach consists of a specification language, called Knot, that allows concise and natural specifications of abstract syntax of programming languages together with their scoping rules and of semantic relations on top of the syntax. We complement Knot with a code generator, called Needle, that specializes the generic definitions and lemmas for the variable binding boilerplate and allows manual customization and extension.

5.3.1 Scientific Output

The first stage in the development of Knot and Needle concerned itself with the abstract syntax only and was published in the article

> Keuchel, S., Weirich, S., and Schrijvers, T. (2016). Needle & Knot: Binder Boilerplate Tied Up. In *Programming Languages and Systems: 25th European Symposium on Programming*, ESOP '16, pages 419–445. Springer.

The framework was subsequently extended with the support for inductive relations, which also includes the symbolic expressions. This part is contained in the article

> Keuchel, S., Schrijvers, T., and Weirich, S. (2016). Needle & Knot: Boilerplate Bound Tighter. Unpublished draft.

The remainder of the second part of this thesis presents the contents of the two articles.

Chapter 6

The KNOT Specification Language

This chapter presents KNOT, a language for specifying programming languages. The language semantics, elaboration of boilerplate and the implementation are discussed in later chapters. We introduce KNOT by example first in Section 6.1 and formally in Sections 6.3, 6.4 and 6.5.

Section 6.3 deals with the specification of *abstract syntax* of programmning languages and their scoping rules. In Section 6.4 we look at *symbolic expressions* that are used in the specification of *inductive relations*. The latter are presented in Section 6.5.

We discuss design choices and ensuing restrictions in Section 6.6 and conclude with our contributions in Section 6.8.

6.1 Knot by Example

In this section we showcase KNOT by porting the semi-formal specification of their $F_{\exists,\times}$ calculus from Chapter 5. Section 6.1.1 discusses the KNOT specification of the abstract syntax of $F_{\exists,\times}$ and Section 6.1.2 its typing relation.

6.1.1 Abstract Syntax Specifications

Figure 6.1 contains the KNOT specification of $F_{\exists,\times}$'s abstract syntax, which corresponds to EBNF grammar specification in Figure 5.1.

```
namespace Tyv : Ty
namespace Tmv : Tm

sort Ty :=                                  sort Pat :=
 + tvar (X @ Tyv)                            | pvar (x : Tmv)
 | tarr (T1 : Ty) (T2 : Ty)                  | ppair (p1 : Pat) ([bind p1]p2 : Pat)
 | tall (X : Tyv) ([X]T : Ty)               fun bind : Pat → [Tmv] :=
 | tprod (T1 : Ty) (T2 : Ty)                 | pvar x → x
 | texist (X : Tyv) ([X]T : Ty)              | pprod p1 p2 → bind p1, bind p2
sort Tm :=
 + var (x @ Tmv)
 | abs (x : Tmv) (T : Ty) ([x]t : Tm)
 | app (t1 : Tm) (t2 : Tm)
 | tabs (X : Tyv) ([X]t : Tm)
 | tapp (t : Tm) (T : Ty)
 | pair (t1 : Tm) (t2 : Tm)
 | case (t1 : Tm) (p : Pat) ([bind p]t2 : Tm)
 | pack (T1 : Ty) (t : Tm) (T2 : Ty)
 | unpack (t1 : Tm) (X : Tyv) (x : Tmv) ([X, x]t2 : Tm)

env Env :=
 + empty
 | evar : Tmv -> Ty : Typing
 | etvar : Tyv ->
```

Figure 6.1: KNOT specification of $F_{\exists,\times}$ (part 1)

While it is not apparent in the EBNF grammar in Figure 5.1, the two sorts for variables and the one for typing contexts have a special purpose that is related to variable binding. KNOT makes the distinction between these and the other sorts explicit and uses different declarations forms to introduce them. Specifically, KNOT distinguishes between *namespaces*, *(regular syntactic) sorts* and *environments* which we discuss in turn.

Namespaces Figure 6.1 starts with the declaration of two namespaces. The line `namespace` *Tyv* : *Ty* introduces the namespace *Tyv* (short for type variables) and declares that it is a namespace for the sort *Ty*, which represents $F_{\exists,\times}$ types and which is defined elsewhere in the figure. Similarly, we declare

Tmv to be a namespace for terms *Tm*.

Regular Syntactic Sorts Three sorts are introduced next: types *Ty*, terms *Tm* and patterns *Pat* using an established notation in functional programming for algebraic datatype declarations. Each sort is defined by a list of constructors of which there are two kinds: *variable constructors* and *regular constructors*.

Variable constructors are introduced with a plus sign. In the example, the line

$$+\, tvar\ (X @ Tyv)$$

declares the variable constructor *tvar* for types. It holds a single *variable reference* of the namespace *Tyv* for type variables.

Regular constructors are declared using the vertical bar and can have an arbitrary number of fields. The line

$$|\ tall\ (X : Tyv)\ ([X]\ T : Ty)$$

declares the regular constructor *tall*, which represents universally quantified types. All fields are explicitly named. The first field declaration $(X : Tyv)$ introduces the field (named) X, which is a binding for a variable of namespace *Tyv*. The second field declaration $([X]\ T : Ty)$ introduces the field T for a subterm of sort *Ty*. It is prefixed by the *binding specification* $[X]$ which stipulates that X is brought into scope in the subterm T. This is exactly the essential scoping information that we highlighted in Figure 5.2. In contrast to Figure 5.2 we do not explicitly model (the domain of) the typing context; all variables that are in scope at the point of the *tall* constructor are implicitly also in scope in all subterms.

Multiple variables can be brought into scope together. For example, the binding specification for the body t_2 of the *unpack* constructor brings both the type variable X and the term variable x into scope.

Binders The sort *Pat* for patterns is special in the sense that it represents a sort of binders. The function *bind* specifies which variables are bound by a pattern, similar to the bnd$(\cdot)$ function in Figure 5.3. The function declaration for *bind* in Figure 6.1 consists of a signature and a body. The signature specifies that patterns bind variables of namespace *Tmv*, and the body defines *bind* by means of an *exhaustive one-level pattern* match. Functions can be used in binding specifications. The term constructor *case* for nested pattern matching uses *bind* to specify that the variables bound by the pattern p are simultaneously brought into scope in the body t_2.

The constructor *ppair* also uses *bind* in a binding specification of the right component, even though patterns themselves do not contain terms or term variable references. However, since *bind concatenates* the bound variables of p_1 and p_2, the binding specification denotes that the variables of p_2 are considered bound after the variables of p_1. This is used in the scope checking of KNOT specifications which is explained in Section 6.3.1.

Environments The last declaration defines typing environments. The plus sign indicates the base case with constructor *empty*. All other cases associate information with variables of a namespace. The constructor *evar* declares that it represents a mapping of term variables *Tmv* to types *Ty*. It also states that the term variable clause is substitutable for judgements of the typing relation *Typing*. We discuss this below where we define *Typing*. The constructor *etvar* is not associating any information with type variables.

6.1.2 Inductive Relation Specifications

Figure 6.2 contains the second part of KNOT specification for $F_{\exists,\times}$: the typing relations *Typing* for terms and *PTyping* for patterns.

The first line of a *relation declaration* fixes the signature of a relation. For *Typing*, the declaration stipulates that it makes use of the typing environment *Env* and has two indices: terms *Tm* and types *Ty*. The remainder of a relation declaration consists of rules. Like for sorts, the are two kinds of rules for relations: *variable rules* and, *regular rules*. Both kinds use notation commonly found for generalized algebraic data-types.

Variable rules Similarly to the abstract syntax, the *variable rules* are introduced with a plus sign. Parameters in braces define lookups, e.g. the parameter $\{x \to T\}$ of *Tvar* represents a lookup of the term variable x in the *implicit typing environment*. We require that each variable rule consists of exactly one lookup which corresponds exactly to the signature of the relation that is being defined and is consistent with the declaration of the environment.

Regular rules The *regular rule Tabs* specifies the typing of term abstractions. In square brackets before a field, we can add *rule binding specifications* that allow us to change the implicit environment for this field. In this case, we extend the implicit typing context with a binding for the λ-bound variable x.

In this rule, the domain type T_2 changes scope. In the semi-formal typing relation in Figure 5.5, the meta-variable τ, that corresponds to T_2, appears

```
relation [Env] Typing Tm Ty :=
 + Tvar : {x → T} → Typing (var x) T
 | Tabs : [x → T1] Typing t (weaken T2 x) →
     Typing (abs x T1 t) (tarr T1 T2)
 | Tapp : Typing t1 (tarr T1 T2) → Typing t2 T1 → Typing (app t1 t2) T2
 | Ttabs : [X →] Typing t T → Typing (tabs X t) (tall X T)
 | Ttapp : Typing t1 (tall X T12) → Typing (tapp t1 T2) (subst X T2 T12)
 | Tpack : Typing t2 (subst X U T2) →
     Typing (pack U t2 (texist X T2)) (texist X T2)
 | Tunpack : Typing t1 (texist X T12) →
     [X →, x → T12] Typing t2 (weaken T2 [X, x]) →
     Typing (unpack t1 X x t2) T2
 | Tpair : Typing t1 T1 → Typing t2 T2 → Typing (prod t1 t2) (tprod T1 T2)
 | Tcase : Typing t1 T1 → (wtp : PTyping p T1) →
     [bind wtp] Typing t2 (weaken T2 (bind p)) → Typing (case t1 p t2) T2
 relation [Env] PTyping Pat Ty :=
 | Pvar : PTyping (pvar x) T; bind = x → T
 | Pprod : (wtp1 : PTyping (pvar x) T1) →
     (wtp2 : [bind wtp1] PTyping p2 (weaken T2 (bind p1))) →
     PTyping (ppair p1 p2) (tprod T1 T2);
       bind = bind wtp1, bind wtp2
```

Figure 6.2: Typing relation for $F_{\exists,\times}$

in the context $\Gamma, y : \sigma$ in the premise and in Γ in the conclusion. The scope change in the semi-formal development is implicit and is only possible because of language specific (non-)subordination information, i.e. the term variable y cannot appear in the type τ. In KNOT the scope change has to be explicitly indicated by weakening T_2 in the premise. Similarly, in the semi-formal development, the subordination information is insufficient for σ in the TUNPACK rule. The rule uses the additional side-condition $\alpha \notin \mathrm{fv}(\sigma)$ to allow σ to be well-scoped in both $\Gamma, \alpha, x : \tau$ and Γ. In the KNOT specification, the T_2 is explicitly weakened by the type variable X, which corresponding to α, and the term variable x. In contrast, in the rule *Ttabs* the body of the universal quantification is under a binder in the conclusion and it does not change its scope so no weakening is performed.

The rule for type applications *Ttapp* shows the use of symbolic substitution (*subst X T2 T12*) in the conclusion and the rule *Tpack* for packing existentials shows symbolic substitution in the premise. Finally, in the rule *Tunpack* we

need to weaken the type T_2 explicitly with the type variable X and the term variable x for the typing judgement of the body t_2.

Relation outputs The typing of patterns can be similarly translated from the semi-formal specification in Section 5.1.2. The additional concern is the definition of the relation output that defines the typing context extension for the variables bound by the pattern, or more precisely defined to be bound by the *bind* function on patterns. In Figure 6.2 this output is explicitly referred to by reusing the function name *bind*. After each rule, we include a clause that defines *bind* for this rule. For this purpose, we also allow naming the fields of rules. Calling *bind* on a field gives access to its output.

6.2 Key Design Choices

This section discusses concepts, that influenced the design of KNOT, which makes it easier to understand the specification of KNOT in the next section and motivate some of the made choices. Section 6.2.1 explain requirements that KNOT puts on sorts to guarantee that boilerplate lemmas for them can be automatically generated. KNOT has two different kinds of meta-variables. The intuition behind them and their treatment are discussed in Section 6.2.2. Section 6.2.3 extends the requirements of Section 6.2.1 from sorts to relations to ensure that their boilerplate is derivable.

6.2.1 Free Monadic Presentations

One of the remaining questions is for which class of languages is the substitution boilerplate derivable? To find the answer to this question, observe that it is folklore that the syntax of lambda calculi has a monadic structure: We can for example model well-scoped terms of the untyped lambda calculus as an ordinary monad on sets using nested datatypes [Bird and Paterson, 1999; Altenkirch and Reus, 1999], or well-scoped and well-typed terms of the simply-typed lambda calculus using a generalization of monads [Altenkirch and Reus, 1999; Altenkirch et al., 2010, 2014]. In these cases, the variable constructor represents the *unit* (also called return) of the monad and (simultaneous) substitution of all variables the *bind*.

However, the syntax of lambda calculi not only has a monadic structure, but it even has a structure that is similar to *free monads*. This is very fortunate since the monadic operations of free monads are derivable from a base functor.

We use this in the design of Knot and require that all sorts with variables follow this structure to make substitutions generically derivable.

We will briefly revise the free monads and their generic construction in Haskell. Subsequently, we present a Haskell definition of well-scoped terms of the untyped lambda calculus and relate it to the generic definition of free monads. Finally, we discuss the design implications for the Knot specification language.

Free Monads on Sets in Haskell

Consider the monad of leafy binary trees *Tree*.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)

instance Monad Tree where
  return          = Leaf
  Leaf a  >>= m = m a
  Fork l r >>= m = Fork (l >>= m) (r >>= m)
```

The return of the monad is given by constructing a leaf from a value and the monadic bind replaces a leaf by a new tree depending on which value the leaf contained. Or put differently, the bind encodes a simultaneous substitution of leaves by trees. The tree datatype has a particular structure: The *Leaf* constructor that forms the return of the monad is the only one that contains a value of the parameter type. Such monads are *free monads* and can be generically constructed from a base pattern functor.

Figure 6.3 shows the generic free monad construction. $Free_{Set}$ constructs a free monad for a given pattern functor. The effect on variables is completely defined in the instance and we only need to traverse f-structures to get to the variable cases. Consequently, the *Monad* instance only requires functoriality of f.

We can then alternatively define *Tree* using $Free_{Set}$ and inherit the generic *Monad* instance for free:

```
type Tree' = Free (,)
```

Well-scoped Lambda Terms

We can define the well-scoped terms of the untyped lambda calculus using generalized algebraic datatypes (GADTs). This representation is also known as *intrinsically well-scoped de Bruijn terms*[Benton et al., 2012]. Figure 6.4 shows the construction. We use natural numbers to denote the number of variables that are in scope and bounded naturals *Fin d* to represent variables. In essence, *Fin d* corresponds to the well-scopedness predicates on variables

```
data Free_Set f a where
  Return_Set ::   a → Free_Set f a
  Step_Set   ::   f (Free_Set f a) → Free_Set f a
instance Functor f ⇒ Monad (Free_Set f) where
  return = Return_Set
  t >>= f = case t of
    Return_Set a → f a
    Free_Set x   → Free_Set (fmap (>>=f) x)
```

Figure 6.3: Free Monads in Haskell

```
data Nat = Z | S Nat
data Fin (d :: Nat) where
  FZ :: Fin (S d)
  FS :: Fin d → Fin (S d)
data Lam (d :: Nat) where
  Var :: Fin d → Lam d
  App :: Lam d → Lam d → Lam d
  Abs :: Lam (S d) → Lam d
substLam :: Lam d1 → (Fin d1 → Lam d2) → Lam d2
substLam (Var x)     m = m x
substLam (App t1 t2) m = App (substLam t1 m) (substLam t2 m)
substLam (Abs t)     m = Abs (substLam t (upSubLam m))
upSubLam :: (Fin d1 → Lam d2) → (Fin (S d1) → Lam (S d2))
upSubLam m FZ     = Var FZ
upSubLam m (FS x) = substLam (m x) (Var ∘ FS)
```

Figure 6.4: Intrinsically Well-Scoped de Brujn terms

$d \vdash^{\text{var}}_{\text{tm}} n$ in Figure 5.8 with the term-level index n removed. The *Lam* type encodes the well-scoped lambda terms. The parameter d encodes the free variables that can potentially appear. In the case of an abstraction its incremented to account for the new lambda-bound variable. Figure 6.4 also defined a simultaneous substitution operator $subst_{Lam}$. A substitution is represented using functions of type $Fin\ d_1 \to Lam\ d_2$ that substitutes all d_1-variables by lambda terms with free variables in d_2. When going under a lambda binder during the substitution needs to be adjusted for the introduced variable which

is handled by $upSub_{Lam}$ [1]. *Lam* together with its simultaneous substitutions form a Kleisli triple in the sense of [Altenkirch and Reus, 1999] and a monad relative on *Fin* in the sense of [Altenkirch et al., 2010, 2014].

Notice, that similar to the bind of the *Tree* datatype, the substitution operator of *Lam* only applies the substitution in the variable case and otherwise passes it through unchanged or extended. This suggests, that we may copy the construction of a free monad on sets to functors on $Nat \to Set$. This generic construction[2] and a generic definition of simultaneous substitution can be found in Section A.1 of the appendix. Also shown in the appendix is an instantiation of *Lam* from a base functor.

Design Implications

We want substitution boilerplate to be derivable for all KNOT specifications and hence apply the insights of this section by enforcing a free monadic shape on syntactic sorts with variables. As already indicated in the example specification in Section 6.1, we always treat the variable case separately from the other cases. Furthermore, we require that there is exactly one one distinguished *variable constructor* per namespace which has a *reference occurrence* as its only argument. All other constructors only contain *binding occurrences* and subterms. This rules out languages for normal forms, but as they require custom behavior (renormalization) during substitution [Sabry and Felleisen, 1993; Watkins et al., 2004] their substitution boilerplate cannot be derived generically anyway.

Our interpretation of KNOT specifications using de Bruijn terms and our implementations differ from the presentation in this Section. We use traditional algrbraic datatypes for our term representation and extrinsic well-scoping predicates instead of the intrinsically well-scoped representation of this section. Both representations are equivalent and this choice does not impact the derivability of the boilerplate. We use single place shifting and substitution instead of simultaneous renaming and substitution to make the generalization to multiple namespaces easier. This is not a restriction since all (well-scoped) simultaneous substitutions can be written as a sequence of single place shifting and substitution [Schäfer et al., 2015a; Keuchel, 2016].

[1] See [Altenkirch and Reus, 1999] for a termination argument for the mutually recursively defined $subst_{Lam}$ and $upSub_{Lam}$.

[2] We have not shown that this construction is indeed free in any formal sense.

6.2.2 Local and Global Variables

In the variable rule of $F_{\exists,\times}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{tm}} x : \tau}\ \textsc{TVar}$$

the variable x is used as a reference and is bound in the context Γ.

On the other hand, in the judgement

$$\Gamma \vdash_{\text{tm}} (\lambda y : \tau.y) : \tau \to \tau.$$

the variable y appears in both, a binding position and a reference position. The reference use of y has to refer to the enclosing binding and not to a binding in the context Γ.

Following the literature on *locally nameless* [Aydemir et al., 2008] and *locally named* [Sato and Pollack, 2010] representations we call y a *locally bound* variable (aka locally scoped variables [Pitts et al., 2015]), or more concisely a *local variable*, and x a *global* or *free variable*.

The distinction between local and global variables goes back to at least Frege [1879] and representations such as locally nameless and locally named have internalized this distinction. These concepts do not commit us to a particular representation of variable binding, such as a locally nameless representation. Rather, these notions arise naturally in meta-languages.

Frege characterizes global variables as variables that can possibly stand for anything while local variables stand for something very specific. Indeed, the variable rule is parameterized over the global (meta-)variable which can refer to any variable in the typing context. As previously mentioned, y can only possibly refer to the enclosing binder. This distinction is also visible in the de Bruijn representation: The variable rule is parameterized over an index for variable x. A local reference, however, is always statically determined. For instance, the index for y in the judgement above is necessarily 0.

The type-substitutions in the rules TTApp for type-application and TPack for packing existential types operate on local variables only. For reasons, that are explained in Section 6.2.3 below, we enforce substitutions in the definition of relations to only operate on local variables.

We adopt the Barendregt variable convention in Knot at the meta-level. Two locally bound meta-variables that have distinct names are considered to represent distinct object-variables, or, put differently, distinct local variables cannot be aliased. However, global meta-variables with distinct names can be aliased, i.e. represent the same object-variable.

6.2.3 Context Parametricity

The variable rule is special in the sense that it is the only rule where a global variable is used. The variable rule performs a lookup of the type in the implicit typing context. More generally, KNOT implicitly assumes that any global variable, independent of an explicit lookup, is bound in the context. As a consequence, the use of a global variable inspects the context.

We call a rule *not context parametric*, iff it makes any assumptions about the context, e.g. through inspection with a global variable. The variable rule of $F_{\exists,\times}$'s typing relation is the only not context parametric rule. The other rules either pass the context through unchanged to the premises, or pass an extended context to the premises without inspecting the prefix. We call these rules *context parametric*.

Context parametricity is important for the automatic derivation of boilerplate. For instance, for the semi-formal substitution lemma of $F_{\exists,\times}$'s typing relation in Section 5.1.3, the inductive step of each regular rule consists of applying the same rule again modulo commutation of substitutions. Independent of the language at hand, this is automatically possible for any context parametric rule.

To understand this, note that the substitution lemma encodes a context change that states the preservation of typing when substituting a global variable. Context parametric rules do not make assumptions about the context, hence they are compatible with any changes to the context as long as the change can be properly reflected in the indices. This raises the question whether this is always possible, even if the rule uses syntactic operations, like for example the type application rule of $F_{\exists,\times}$ which uses a type substitution and hence requires the commutation of two substitutions. However, this will always be a substitution of a global variable, i.e. our context change, and a local variable substitution. Intuitively, such a commutation is always possible.

6.3 Knot Syntax

Figure 6.5 shows the grammar of KNOT. A KNOT language specification *spec* consists of variable namespace declarations *namedecl*, syntactic sort declarations *sortdecl*, function declarations *fundecl*, environment declarations *envdecl* and relation declarations *reldecl*. We defer explaining relation declarations until Section 6.5.

A namespace declaration `namespace` $\alpha : S$ introduces the namespace α and associates it with the syntactic sort S. This expresses that variables of names-

Labels

α, β, γ	*Namespace label*	s, t	*Sort meta-variable*
b	*Binding meta-variable*	f	*Function label*
g	*Global meta-variable*	E	*Env label*
S, T	*Sort label*	R	*Relation label*
K	*Constructor label*	r	*Rule label*

Declarations and definitions

spec	::=	$\overline{decl}$	*Specification*
decl	::=	*namedecl* | *sortdecl* | *fundecl*	*Declaration*
	|	*envdecl* | *reldecl*	
namedecl	::=	`namespace` $\alpha : S$	*Namespace*
sortdecl	::=	`sort` S `:=` $\overline{condecl}$	*Sort*
condecl	::=	`+`K `(`g `@` α`)`	*Constr. decl.*
	|	`|`K $\overline{(b : \alpha)}$ $\overline{(\texttt{[}bs\texttt{]}s : S)}$	
bs	::=	$\overline{bsi}$	*Binding spec.*
bsi	::=	b | fs	*Bind. spec. item*
fundecl	::=	`fun` $f : S$ `->` $[\overline{\alpha}]$ `:=` $\overline{funclause}$	*Function*
funclause	::=	$K\, \overline{b}\, \overline{s}$ `->` bs	*Function clause*
envdecl	::=	`env` E `:=` $\overline{envclause}$	*Environment*
envclause	::=	`+`K	*Empty env.*
	|	`|`$K : \alpha$ `->` $\overline{S} : R$	*Env. clause*

Figure 6.5: The Syntax of KNOT

pace α can be substituted for terms of sort S. While most languages feature at most one namespace per sort, it is nevertheless possible to associate multiple namespaces with a single sort. This can be used, e.g., in languages with linear type systems to distinguish linearly bound from unrestricted variables.

A declaration of sort S comes with two kinds of constructor declarations *condecl*. Variable constructors `+`K `(`g `@` α`)` hold a variable reference g in the namespace α. These are the only constructors where variables are used as references. The global variable reference g signifies that the reference is free when considering a variable constructor in isolation. In larger symbolic expressions, also binding variables may appear in variable constructors.

Regular constructors $K\, \overline{(b : \alpha)}\, \overline{(s : S)}$ contain named variable bindings $\overline{(b : \alpha)}$ and named subterms $\overline{(s : S)}$. For the sake of presentation we assume that the

variable bindings precede subterms.

Every subterm s is preceded by a binding specification bs that stipulates which variable bindings are brought in scope of s. The binding specification consists of a list of items bsi. An item is either a singleton variable binding b of the constructor or the invocation of a function f, that computes which variables in siblings or the same subterm are brought in scope of s. Functions serve in particular to specify multi-binders in binding specifications. In regular programming languages the binding specifications of most subterms are empty; to avoid clutter we omit empty binding specifications `[]` in the concrete syntax of KNOT.

Functions are defined by function declarations *fundecl*. The type signature $f : S \to [\overline{\alpha}]$ denotes that function f operates on terms of sort S and yields variables in namespaces $\overline{\alpha}$. The function itself is defined by exhaustive case analysis on a term of sort S. A crucial requirement is that functions cannot be defined for sorts that have variables. Otherwise it would be possible to change the set of variables that a pattern binds with a substitution. The specification of the output type $\overline{\alpha}$ is used by NEEDLE to derive *subordination-based strengthening* lemmas [Keuchel et al., 2016]. For simplicity we ignore the output type of functions and any other subordination related information in the remainder of this thesis.

Environments E represent a mapping from variables in scope to additional data such as typing information. To this end, an environment declaration *envdecl* consists of named clauses $K : (\alpha \texttt{ -> } \overline{S} : R)$ that stipulate that variables in namespace α are mapped to terms of sorts $\overline{S}$. Additionally, we specify that this clause can be substituted for judgement of relation R. If the relation R is omitted, then it defaults to well-scopedness of the data. We elaborate on this, together with the syntax of inductive relations, in Section 6.5.

6.3.1 Well-Formed Knot Specifications

This section defines which KNOT specifications are well-formed. To simplify the explanation of well-formedness and of the semantics of KNOT specifications, we disregard both function declarations and only consider single-variable binding for the rest of this section and the following. See the technical appendix for the extended formalization.

Figure 6.6 defines the well-formedness relation $\vdash$ *spec* for KNOT specifications. The single rule WFSPEC expresses that a specification is well-formed if each of the constructor declarations inside the sort declarations is and the meta-environment $\mathcal{V}$ contains exactly the declared namespaces.

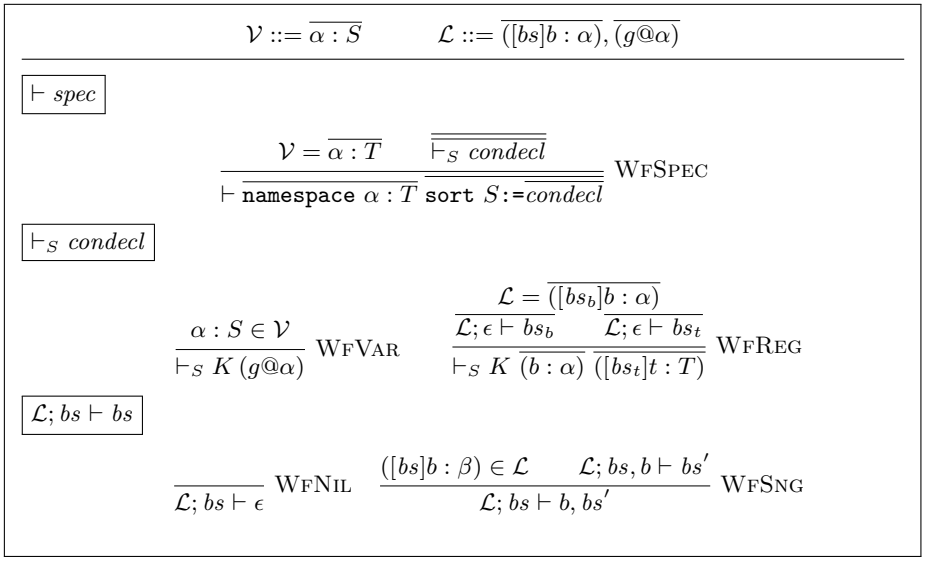


Figure 6.6: Well-formed specifications

The auxiliary well-sorting relation $\vdash_S$ *condecl* denotes that constructor declaration *condecl* has sort S. There is one rule for each constructor form. Rule WfVar requires that the associated sort of the variable namespace matches the sort of the constructor. Rule WfReg handles regular constructors. It builds a constructor-local meta-environment $\mathcal{L}$ for binding fields $([bs_b]b : \alpha)$. The binding specification bs_b of a binding b denotes the *local scope* into which the corresponding object-variable is introduced. The local scope is left implicit in the syntax; hence, it needs to be inferred in this rule. The binding specifications of fields are checked against $\mathcal{L}$. Also, we check clauses of function declarations as part of this rule. We use the notation $f\ (K\ \overline{b'}\ \overline{t'}) = bs'$ to look up the clause of f for constructor K. After proper renaming, the right-hand side of each functional clause has to be consistent with $\mathcal{L}$.

The relation $\mathcal{L}; bs_1 \vdash bs_2$ in Figure 6.6 denotes that binding specification bs_2 is well-formed with respect to the scope bs_1. The relation ensures that the order of different binding items has is consistent across all binding specifications and there are no gaps. For instance, if one of the binding specifications is $[b_0, b_1, b_2]$ then another field of the same constructor cannot have the binding specification $[b_0, b_2, b_1]$ or $[b_0, b_2]$. This restriction prevents the user from relying on a structural exchange property of environments when specifying

$$\begin{array}{rcll} sym & ::= & s \mid K\ \overline{b}\ \overline{sym} \mid \textbf{weaken}\ sym\ bs & \textit{Symbolic exp.} \\ & \mid & K\ g \mid K\ b \mid \textbf{subst}\ b\ sym\ sym & \end{array}$$

$\boxed{\mathcal{L}; bs \vdash sym : S}$

$$\frac{K : (\overline{[bs_b]b : \alpha}) \to (\overline{[bs_t]t : T}) \to S \quad (\![\{b \mapsto b'\}bs_b]b' : \alpha) \in \mathcal{L} \quad \overline{\mathcal{L}; bs, \{b \mapsto b'\}bs_t \vdash sym : T}}{\mathcal{L}; bs \vdash K\ \overline{b'}\ \overline{sym} : S}\ \textsc{SymReg}$$

$$\frac{[bs]s : S \in \mathcal{L}}{\mathcal{L}; bs \vdash s : S}\ \textsc{SymVar} \qquad \frac{K : \alpha \to S \quad (g@\alpha) \in \mathcal{L}}{\mathcal{L}; bs \vdash K\ g : S}\ \textsc{SymGbl} \qquad \frac{K : \alpha \to S \quad ([bs]b : \alpha) \in \mathcal{L}}{\mathcal{L}; bs, b, bs' \vdash K\ b : S}\ \textsc{SymLcl}$$

$$\frac{\mathcal{L}; bs \vdash sym : S}{\mathcal{L}; bs, bs' \vdash \textbf{weaken}\ sym\ bs' : S}\ \textsc{SymWeaken}$$

$$\frac{([bs]x : \alpha) \in \mathcal{L} \quad (\alpha : T) \in \mathcal{V} \quad \mathcal{L}; bs \vdash sym_1 : T \quad \mathcal{L}; bs, x \vdash sym_2 : S}{\mathcal{L}; bs \vdash \textbf{subst}\ x\ sym_1\ sym_2 : S}\ \textsc{SymSubst}$$

Figure 6.7: Symbolic expressions and their well-formedness

inductive relations which in turn enables us to deal with environment well-scopedness generically in the derivation of judgement well-scopedness lemmas.

Rule WfNil regulates the base case of an empty binding specification that is always well-scoped. By rule WfSng a singleton binding is well-scoped if the local scope bs is consistent with the information in the local environment $\mathcal{L}$ and it checks the tail bs' in the extended scope bs, b.

Including function calls in the binding specification requires checking them for well-scopedness too which can be found in Appendix A. In short: For calling a function $(f : T \to \overline{\alpha})$ on a field $([bs]t : T)$, we require $\mathcal{L}; bs \vdash f\ t$, i.e. the local scope of the function call is the binding specification of s. However, this is very restrictive in general since it rules out scoping constructs such as recursive scoping. We come back to this issue in the concluding discussion of this chapter in Section 6.6.

6.4 Symbolic Expressions

This section defines *symbolic expressions* on top of specification declarations. These are needed for the declaration of inductive relations on sorts. The general idea is that we extend sort terms with meta-variables and with symbolic constructs for meta-operations such as substitution. These meta-variables are distinct from the object-language variables. We can for example have a meta-variable for a term of a sort that has no namespaces.

Figure 6.7 (top) contains the grammar for symbolic expressions. An expression is a meta-variable s or a regular constructor applied to variable bindings and other symbolic expressions ($K\ \overline{b}\ \overline{sym}$). For variable constructors we need to make a distinction between global ($K\ g$) and local references ($K\ b$). Furthermore, a symbolic expression can also be a reified substitution (**subst** $b\ sym_1\ sym_2$), that denotes a substitution of sym_1 for b in sym_2. We only allow substitution of locally bound variables to ensure context parametricity. The last expression former is a reified weakening (**weaken** $sym\ bs$) that makes context changes explicit. For example consider η-reduction for $F_{\exists,\times}$:

$$abs\ x\ T\ (app\ (weaken\ t\ x)\ (var\ x)) \longrightarrow_\eta t.$$

Here the term t is assumed to be in the outer context of the whole expression and is explicitly weakened under the abstraction. The symbolic weakening implies and replaces freshness conditions. We discuss larger examples of symbolic expressions after introducing inductive relations in Section 6.5.

6.4.1 Expression Well-formedness

When using symbolic expressions we also want to ensure that these are well-sorted and well-scoped with respect to the specification and scoping rules that are defined by the binding specifications of the sorts. Symbolic expressions can themselves introduce new bindings and local references have to be checked to be locally bound. Therefore, we need to keep track of all local bindings that are in scope. We reuse the representation of binding specifications bs to also represent *local scopes*.

The checking is complicated by the fact that arbitrary expressions may appear in a term constructor that contains a binding specification with function calls. So to define well-scopedness of expressions, we first have to define symbolic evaluation of functions on expressions. This evaluation normalizes function calls $f\ sym$ down to ordinary binding specifications that only contain function calls on meta-variables $f\ s$. During evaluation we need to pattern match regular term constructions against function clauses. This pattern

j	::=	*Judgement var.*
reldecl	::= `relation [`E`]` $R\,\overline{S}$:= $\overline{ruledecl}$	*Relation decl.*
	| `relation` $R\,\overline{S}$:= $\overline{ruledecl}$	
ruledecl	::= `+`r : *lookup* `->` *jmt* | `|`r : $\overline{fml}$ `->` *jmt*; $\overline{f = rbs}$	*Rule decl.*
fml	::= *lookup* | `[`*rbs*`]` j : *jmt*	*Formula*
lookup	::= {x `->` $\overline{sym}$}	*Lookup*
jmt	::= R $\overline{sym}$	*Judgement*
rbs	::= $\overline{rbsi}$	*Rule binding spec.*
rbsi	::= b `->` $\overline{sym}$ | f j	*Rule bind. spec. item*

Figure 6.8: Syntax for relations

matching yields a symbolic environment θ that maps binding meta-variables to new names and sort meta-variables to expressions. Symbolic environments θ are defined in Figure 6.7 (top).

Well-formedness Finally, Figure 6.7 (bottom) shows the definition of well-formedness of symbolic expressions. The relation $\mathcal{L}; bs \vdash sym : S$ denotes that the symbolic expression sym has sort S and is well-formed in scope bs under the local environment $\mathcal{L}$.

The rule SymVar looks up the sort and scope of a meta-variable for a sort term in $\mathcal{L}$. Variable constructors are handled by two rules. Rule SymLcl is used in case the variable is bound locally and bs' represents the difference to the scope of the binding. Global variables are handled by rule SymGbl. The case of a regular constructor is handled by rule SymReg. For each of the fields $[bs_t]t : T$ the binding specification bs_t the corresponding symbolic expression sym is checked in the extended scope $(bs, \{b \mapsto b'\}bs_t)$ where $\{b \mapsto b'\}$ denotes simultaneous renaming of the bindings b to b'. Rule SymWeaken strengthens the scope bs, bs' of a symbolic weakening (**weaken** sym bs'). The symbolic expression sym is checked in the stronger scope bs. Finally, rule SymSubst takes care of single variable substitutions. The expression sym_2 lives in the extended scope bs, b. Hence, only substitution of the last introduced binding is allowed. The sort and scope of the substitute sym_1 have to agree with that of b.

6.5 Inductive Relations

Figure 6.8 shows the grammar for specification of relations. A relation declaration *reldecl* introduces a new relation R with an optional environment index E and indices $\overline{S}$. For the purpose of variable binding, we regard the first sort index to be classified by the remaining ones. The environment E itself is left implicit in the rules; only environment changes are explicitly stated. Each *reldecl* contains a list of named rules r of which there are two kinds. Regular rules $\texttt{|}\, r : \overline{fml}\ \texttt{->}\ jmt; \overline{f = rbs}$ contain a list of formulas as premises and conclude in a judgement which is simply a relation between symbolic expressions. We also allow the definition of function counterparts at the level of relations, but instead of having a separate declaration form, we declare them inline with relations.

A formula is either a variable lookup in the environment, that gives access to the associated data, or a judgement that can be named with judgement variables. Similar to binding specification of sort fields, judgements are prefixed with rule binding specifications *rbs* that alter the implicit environment. These consist of a list of items: either singleton binding variables mapped to associated data $\overline{sym}$ or function calls $(f\ j)$ on judgements. The second kind of rules are variable rules $\texttt{+}r : lookup\ \texttt{->}\ jmt$ that only contain a single lookup as a premise.

Note that allowing lookups in regular rules is a departure from our free-monadic view on syntax. Furthermore, we do not require that variable rules are declared for each environment clause. The reason is that relations that do not fit into this view are quite common, e.g. most algorithmic type systems require renormalization during substitution. Hence, we provide support for these relations and leave proof obligations for the user in order to generate substitution lemmas. Each regular rule that makes use of lookups gives rise to an obligation. If there is no explicit variable rule for an environment clause, the corresponding derived rule needs to be proven.

6.5.1 Relation Well-formedness

Finally, we define the well-formedness of relation specifications in Figure 6.9. We make use of a global meta-environment $\mathcal{R}$ that contains the environment and sort types of relations. The meta-relation $\vdash$ *reldecl* delegates the well-formedness checking of relation declarations to $\vdash_{E?,R,\overline{S}}$ *ruledecl* which checks the individual rules with respect to the given names $E?, R, \overline{S}$. In case of a variable rule RuleVar, the relation needs to have an environment E and the clause for the namespace α of the free variable g needs to be substitutable by

$E_? ::= E \mid \bullet$ Optional Env. $\quad \mathcal{R} ::= \overline{R : E_? \times \overline{S}}$ Relation meta-env.

$\boxed{\vdash \mathit{reldecl}}$

$$\frac{\overline{\vdash_{E,R,\overline{S}} \mathit{ruledecl}}}{\vdash \texttt{relation}\ [E]\ R\ \overline{S} := \overline{\mathit{ruledecl}}}\ \textsc{WfRelEnv}$$

$$\frac{\overline{\vdash_{\bullet,R,\overline{S}} \mathit{ruledecl}}}{\vdash \texttt{relation}\ R\ \overline{S} := \overline{\mathit{ruledecl}}}\ \textsc{WfRelNoEnv}$$

$\boxed{\vdash_{E_?,R,\overline{S}} \mathit{ruledecl}}$

$$\frac{K : \alpha \to S \qquad (K' : \alpha\ \texttt{->}\ \overline{T} : R) \in E}{\vdash_{E,R,(S,\overline{T})} \texttt{+}r : \{g \mapsto \overline{t}\} \to R\ (K\ g)\ \overline{t}}\ \textsc{RuleVar}$$

$$\frac{\overline{\mathcal{L} \vdash_{E_?} \mathit{fml}} \qquad \overline{\mathcal{L}; \epsilon \vdash \mathit{sym} : S}}{\vdash_{E_?,R,\overline{S}} \texttt{|}r : \overline{\mathit{fml}} \to R\ \overline{\mathit{sym}}}\ \textsc{RuleReg}$$

$\boxed{\mathcal{L} \vdash_{E_?} \mathit{fml}}$

$$\frac{(R : E_? \times \overline{T}) \in \mathcal{R} \vee (R : \bullet \times \overline{T}) \in \mathcal{R} \qquad \mathcal{L} \vdash_{E_?} \mathit{rbs} \Downarrow \mathit{bs} \qquad ([\mathit{bs}]j : R\ \overline{\mathit{sym}}) \in \mathcal{L} \qquad \overline{\mathcal{L}; \mathit{bs} \vdash \mathit{sym} : T}}{\mathcal{L} \vdash_{E_?} [\mathit{rbs}]j : R\ \overline{\mathit{sym}}}\ \textsc{FmlJmt}$$

$$\frac{(K' : \alpha\ \texttt{->}\ \overline{T}) \in E \qquad (g@\alpha) \in \mathcal{L} \qquad \overline{\mathcal{L}; \epsilon \vdash \mathit{sym} : T}}{\mathcal{L} \vdash_E \{g \mapsto \overline{\mathit{sym}}\}}\ \textsc{FmlLookup}$$

$\boxed{\mathcal{L} \vdash_{E_?} \mathit{rbs} \Downarrow \mathit{bs}}$

$$\frac{}{\mathcal{L} \vdash_{E_?} \epsilon \Downarrow \epsilon}\ \textsc{RbsNil}$$

$$\frac{\mathcal{L} \vdash_E \mathit{rbs} \Downarrow \mathit{bs} \qquad ([\mathit{bs}]b : \beta) \in \mathcal{L} \qquad (K' : \beta\ \texttt{->}\ \overline{T}) \in E \qquad \overline{\mathcal{L}; \mathit{bs} \vdash \mathit{sym} : T}}{\mathcal{L} \vdash_E \mathit{rbs}, b\ \texttt{->}\ \overline{\mathit{sym}} \Downarrow \mathit{bs}, b}\ \textsc{RbsSng}$$

Figure 6.9: Well-formed relations

the relation R. The relation of the conclusion is R and the first index is the variable constructor for namespace α with the lookup variable. The remaining indices are sort meta-variables and the arity and order is exactly the data of

the lookup. This form ensures that we can always wrap a lookup of the clause in the variable rule.

Regular rules are handled by RuleReg. Again, the relation of the judgement in the conclusion is R. Each regular rule has a local-environment $\mathcal{L}$ that is inferred. We check the well-formedness of the symbolic expression in the conclusion against the empty local scope ϵ. This encodes the assumption that all indices are in the same scope and there is no binding between them. The definitions of the functions are checked by the "flattening" relation $\mathcal{L} \vdash_{E_?} rbs \Downarrow bs$. The output bs is ignored. An additional (implicit) requirement is that r has a function definition for each function that is declared on the R's first index sort.

The well-formedness of the formulas of the premise is delegated to the relation $\mathcal{L} \vdash_{E_?} fml$. For lookups, the rule FmlLookup checks that an environment E is given and that the data of the lookup is well-formed with the sorts of the corresponding clause of E. In case of a judgement, we get the environment and sort types of the judgement's relation R from $\mathcal{R}$. R's environment is either the same as that of the enclosing relation or R does not have an environment. The rule binding specification rbs is flattened to a local scope bs which has to match the scope declared in $\mathcal{L}$. The indices $\overline{sym}$ are checked against bs.

Finally, the flattening relation $\mathcal{L} \vdash_{E_?} rbs \Downarrow bs$ calculates the local scope bs that is induced by a rule binding specification rbs. The nil case is straightforward. In cases of a non-empty rbs we need to have an implicit environment E. Rule RbsSng flattens a singleton rule binding b `->` $\overline{sym}$ to b and checks the symbolic expressions against the prefix bs and the sort types $\overline{T}$ of the environment clause. A function call $f\ j$ is handled by rule RbsCall. Its flattening is symbolic evaluation of f on the first index sym. Also, the local scope bs of j is checked to be identical to the flattening of the prefix rbs.

6.6 Discussion

Not all programming languages and scoping rules can be specified in Knot. Some restrictions are imposed by deliberate simplifications to reduce the complexity while other constructs have not been in the targeted scope of Knot. We discuss several importing restrictions, their relevance and future work to lift these restrictions.

Multiple Input Scopes Specification of type systems often use multiple environments, for instance, to split type variables from term variables, or to

have a separate global environment with information about datatypes or top-level bindings. To simplify recursion and induction KNOT however only allows a single sequentially scoped input environment. At the level of terms this means that there is always exactly one input domain that defines the variables in scope. As a result, the user is forced to always share the environment for all namespaces and has to account for all namespaces when reasoning.

For type safety proofs this is rarely an issue, but it may complicate other meta-theoretic proofs. For instance, this unnecessarily burdens the proof of strong normalization of System F via logic relations [Girard et al., 1989].

There is nothing that fundamentally forbids us to support multiple scopes in the future and also allow recursively instead of sequentially scoped global environments. The recursion over terms and relations then however depends on the dependency structure between multiple scopes, which adds a new dimension of complexity into the boilerplate elaboration.

Scope Delimitation Once names are introduced, they are universally visible in all sub-terms[3]. Some languages do however limit the visibility. For instance, imperative languages often allow labels on loops which can be used as the target of a break or continue statement. Generally, these labels are only visible inside the same function, but for example not in the body of any nested functions or lambda expressions. With the support of multiple input scopes, we can encode the scope restriction by for example allowing scopes of subterms to be closed. Another application would be the distinction between stack allocated local variables and heap allocated closure variables that have a similar scoping restriction.

Recursive Scoping The first version of the framework, as presented in [Keuchel et al., 2016], uses a more lenient well-formedness relation for binding specifications than the one presented in Figure 6.6. This alternative version allowed for recursive scoping to be specified using cyclic binding specifications which Figure 6.6 rules out. This is a trade-off between expressivity and simplicity.

To illustrate this, consider a hypothetical typing judgement for a mutual recursive declarations list ds that binds Δ variables with their types. The well-scopedness lemma for terms of such a language require us to proof the following derivation:

[3]Disregarding any kind of shadowing in a nominal interpretation

$$\frac{\Gamma, \Delta \vdash_{\text{decls}} ds : \Delta}{\Gamma, \Delta \vdash ds \wedge \Gamma \vdash \Delta}$$

However, the well-scopedness hypothesis only gives us $\Gamma, \Delta \vdash \Delta$. The usual step is to argue that Δ only has term variable bindings and terms do not appear in typing contexts or in types. Therefore, we can use subordination based strengthening to get $\Gamma, \Delta \vdash \Delta$. KNOT provides enough information to allow NEEDLE to derive such lemmas. However, the cyclicity of such specifications adds unnecessary complexity to the checking and elaboration of symbolic expressions. Furthermore, this approach does not scale to richer type theories that allow inductive-recursive or inductive-inductive declarations, for which the subordination used above is not valid.

Instead of pushing the shortcut over subordination information, we plan to solve the problem in a more principled manner in future work by allowing multiple (potentially circular, but not cyclic) input scopes. Recursive constructs can then be checked with two scopes: one for declaration heads and one for declaration bodies.

Symbolic Substitution Checking the scopes of expressions for language that define scoping functions, requires such functions to be symbolically evaluated on expressions. These definitions can be found in the technical appendix A.

Notably absent from the symbolic evaluation are rules for symbolic substitutions and weakenings. The de Bruijn representation admits for example the rule

$$\frac{f(sym_2) \Downarrow bs'}{f(\mathbf{subst}\ x\ sym_1\ sym_2) \Downarrow bs'}.$$

which expresses that the variables bound by sym_2 remain invariant under substitution and this rule is also used in our elaboration of boilerplate lemmas for the de Bruijn representation (cf. Section 8.1.2). Yet, adding this rule to the specification language would break subject reduction of symbolic evaluation. The reason is that the typing of bs in Figure 6.7 (bottom) is not strong enough to keep track of the scope when performing substitutions or weakenings. In essence, the result cannot be bs' but has to be "bs' without x". Tracking scopes during substitutions or other user-defined functions is the focus of research on *binding safe programming* [Pouillard and Pottier, 2010; Stansifer and Wand, 2014]. In the framework of [Pouillard and Pottier, 2010], bs' in the premise and conclusion of the above rule are two distinct (chains of) weak

links with distinct types, which are in a commutative relationship with the world inclusion induced by the substitution.

We side-step the issue by sticking to the simple scope checking of Figure 6.7 (bottom) and effectively disallow symbolic substitutions and weakenings to appear in positions that are accessed by functions. Another consequence is that substitution and weakening are only allowed "at the end of the context". These restrictions are usually met by relations for typing and operational semantics, but useful examples that violate this restriction exists.

To see this, consider extending the patterns of $F_{\exists,\times}$ with matching on existentials. The adapted sort for patterns may look something like this:

```
sort Pat :=
  | pvar (x : Tmv) (T : Ty)
  | ppair (p1 : Pat) (p2 : Pat)
  | pexist (X : Tyv) ([X]p : Pat)
fun bind : Pat → [ Tmv ] :=
  | pvar x → x
  | pprod p1 p2 → bind p1, bind p2
  | pexist X p → X, bind p2
```

Note that in comparison to Figure 6.1 we have added an additional typing annotation in *pvar* and omitted the binding specification of p_2 in the *ppair* constructor. The latter is necessary to faithfully encode the scoping rules: type-variables of p_1 should not be brought into scope in p_2. However, *bind* still concatenates the recursive results since variables from both sub-patterns should be brought into scope in the body of the match. In relations defined on top of patterns, we can now force write a symbolic expression well-scoped in *bind* p_2 that needs to be transported to scope *bind* p_1, *bind* p_2 which however needs weakening at depth *bind* p_2. The restriction comes already into play in the definition of *bind*: the case for *ppair* is ill-typed.

In future work we would like to extend the scope checking to correctly handle substitutions and weakenings in the middle of the context. Preliminary experimentation suggests that this requires an explicit representation of bindings that have been commuted with substitutions or weakenings and non-trivial equalities that are reminiscent of the equational theory of the σ-calculus [Abadi et al., 1991a]. We also hope to introduce first-class substitutions and develop the scope checking for them.

Heterogeneously Scoped Relations Indices of relations in KNOT are implicitly assumed to be well-scoped in the domain of the implicit environment. For some relations involving binders however, an index may refer to addi-

tional variables bound by another index. Consider a pattern matching relation *Match* p v t_1 t_2 that denotes that successfully matching pattern p against value v and applying the resulting substitutions to t_1 results in t_2. In this case, the variables of p need to be in scope in t_1 which currently is not expressible in KNOT. This is only a moderate extension and should be considered an oversight in the design of KNOT rather than a fundamental problem. Similarly, the classifiers of a variable in an environment are assumed to be homogeneously scoped, which can also be extended.

Computational Functions The biggest restrictions of KNOT lie in the expressivity of symbolic expressions for relation declarations. KNOT only allows constructor applications and symbolic operations. Computational functions can be specified in a relational style. In practice, defining recursive functions is more convenient and supporting computational function calls in expression is very useful. Supporting and scope checking recursive functions and generically deriving their boilerplate is within KNOT's grasp, since the complexity is similar to that of relations. The major difference is that we need to support elimination in addition to introduction forms in expression which come with termination concerns.

Dependent types Constraint on sub-terms are often encoded by using universal quantification, implications and functions on sub-terms. Fundamental support for these constructions is not a problem as long as any syntactic sorts and relations that contain variable cases are only used in positive positions. Unfortunately, this is not always the case. For instance, Vouillon's solution to the POPLMARK challenge [Vouillon, 2012] defines recursive functions for record label lookups and uses these to specify width and depth sub-typing. Specifically, the subtyping rule for records universally quantifies over terms and uses lookups and identity types in negative positions to encode label subsets.

This encoding of a rule poses a major complication for weakening and substitution lemmas. For each of the types used in negative positions, we have to prove that we can undo a substitution. In other words, these types need to have a substitution lemma which is in fact a natural isomorphism.

In summary, extending KNOT in the future with better support for logical primitives is possible but extremely challenging.

6.7 Related Work

Knot grew out of the Ott specification language [Sewell et al., 2010] which allows the specification of *concrete syntax* of languages, including binding specifications, and of inductive relations on terms. However, in contrast to Knot, well-scoping of relations is left unchecked in Ott. The Ott code generator produces code for various proof assistants, including Coq. While Ott only generates datatype and function definitions for syntax and relations, support for lemmas is provided by the additional LNGen [Aydemir and Weirich, 2010] tool which generates syntax-related boilerplate for locally-nameless representations but does not tackle any boilerplate for semantic relations.

Romeo [Stansifer and Wand, 2014] is a programming language that checks for safe handling of variables in programs. Romeo's specification language is based on the concept of attribute grammars [Knuth, 1968] with a single implicit inherited and synthesized attribute. In this view, Knot also has a single implicit inherited attribute and binding specification functions represent synthesized attributes. Moreover, Knot allows multiple functions over the same sort. However, Romeo is a full-fledged programming language while Knot only allows the definition of functions for the purpose of binding specification. Romeo has a deduction system that rules out unsafe usage of binders but is not targeting mechanizations of meta-theory.

Logical frameworks like Abella [Gacek, 2008], Twelf [Pfenning and Schürmann, 1999] and Beluga [Pientka and Dunfield, 2010] are specifically designed to reason about logics and programming languages. Their specialized meta-logic encourages the use of higher-order abstract syntax (HOAS) to represent object-level variable binding with meta-variable bindings, i.e. datatype declarations in these frameworks already serve as specifications of syntax including binding. However, generally only direct support for single variable binding is provided and other binders have to be encoded by transforming the object language. These systems also allow the use of higher-order judgements which is used to model bindings at the level of relations.

6.8 Contributions

Knot contributes several insights into the design of specification languages. The possibility of boilerplate derivation for freely generated syntax is folklore. However, this work extends the idea to relation defined on top of syntax.

A critical ingredient is the identification of local and global variables as part of the meta-language itself instead of being a purely representational

artifact. This notion is for instance not explicitly represented in Ott [Sewell et al., 2010] or in Nominal Isabelle [Urban and Tasson, 2005]. The splitting of these two kind of variables then further allows us to distinguish allows context parametric from non-parametric parametric rules which is the decisive requirement for automatic derivation of substitution lemmas.

Another contribution is the design of a scope-checking system for expressions. This is directly related to binding safe programming. Languages like FreshML [Shinwell et al., 2003] and Romeo [Stansifer and Wand, 2014] can clearly represent and check more sophisticated expressions, but to the best of our knowledge, Knot is the first language to adapt this in a mechanized setting and derive well-scopedness proofs.

Chapter 7

Semantics

The previous chapter has introduced the KNOT specification language for abstract syntax. This chapter generically defines the semantics of KNOT in terms of a de Bruijn representation. We assume a given and fixed well-formed specification *spec* in the rest of this chapter. The discussion follows these consecutive steps:

Section 7.1 declares *de Bruijn syntax terms* for KNOT's *abstract syntax declarations* of the given specification and specifies which are well-sorted and well-scoped with respect to the specification. It also defines the *semantics of binding specifications* by means of evaluation. *Evaluation of expressions* is the topic of Section 7.2. A dependency of evaluation is the definition of boilerplate *shiftings and substitutions function* that operate on terms. These are also defined in Section 7.2. The *interpretation of relation specifications* is discussed in Section 7.3 including boilerplate *environment lookup relations*.

7.1 Syntax terms

This section generically defines the semantics of KNOT in terms of a de Bruijn representation. The goal is to fully define what well-scoped object language terms are with respect to a specification. We therefore start with the definition of de Bruijn terms in Section 7.1.1 and their well-sortedness. Section 7.1.2 follows with the evaluation of binding specification of well-sorted terms. Finally, Section 7.1.3 discusses the well-scopedness judgement for de Bruijn terms.

$$
\begin{array}{llll}
n, m & ::= 0 \mid S\ n & & \text{de Bruijn index} \\
u, v, w & ::= K\ n \mid K\ \overline{u} & & \text{de Bruijn term} \\
h, c & ::= 0 \mid S_\alpha\ h & & \text{Het. number} \\
\vartheta & ::= \overline{g \mapsto n}, \overline{t \mapsto u}, \overline{(f, j) \mapsto u} & & \text{Value env.}
\end{array}
$$

Figure 7.1: KNOT semantics: key definitions

$$\boxed{\vdash u : S}$$

$$\frac{K : \alpha \to S}{\vdash K\ n : S}\ \textsc{SortedVar} \qquad \frac{K : \overline{x : \alpha} \to \overline{[bs]t : T} \to S \quad \vdash u_i : T_i \quad (\forall i)}{\vdash K\ \overline{u} : S}\ \textsc{SortedCtor}$$

$$\boxed{\vdash u : E}$$

$$\frac{K : E}{\vdash K : E}\ \textsc{SortedNil} \qquad \frac{K : E \to \alpha \to \overline{T} \to E \quad \vdash v : E \quad \vdash u_i : T_i \quad (\forall i)}{\vdash K\ v\ \overline{u} : E}\ \textsc{SortedCons}$$

Figure 7.2: Well-sortedness of terms

7.1.1 Raw Terms

Figure 7.1 contains a term grammar for raw terms u. A de Bruijn term consists of either a term constructor applied to a de Bruijn index or a term constructor applied to other terms, which model KNOT's *variable* respectively *regular constructors*. Figure 7.2 contains the well-sortedness relations $\vdash u : S$ for raw sort terms and $\vdash u : E$ for raw environment terms.

There are two rules for sort terms both of which check the sort information given by the specification. Note that in the rule for regular constructors, the names of bindings are dropped in the representation. There are also two rules for environment terms, one for an empty environment and one for extending an environment with a new association. In the latter, the binding is dropped from the representation.

7.1.2 Binding Specification Evaluation

The binding specification $[bs]\ t$ for a particular subterm t of a given term constructor K defines the variables that are brought into scope in t. These

$$\boxed{[\![\ \_\ ]\!]_{\_} : bs \to \vartheta \to h}$$

$$\begin{aligned}
[\![\ \epsilon\ ]\!]_{\vartheta} &= 0 \\
[\![\ bs, x_\alpha\ ]\!]_{\vartheta} &= [\![\ bs\ ]\!]_{\vartheta} + 1_\alpha \\
[\![\ bs, f\ t\ ]\!]_{\vartheta} &= [\![\ bs\ ]\!]_{\vartheta} + [\![\ f\ ]\!](\vartheta\ t)
\end{aligned}$$

$$\boxed{[\![\ \_\ ]\!](\_) : f \to u \to h}$$

$$[\![\ f\ ]\!](K\ \overline{u}) = [\![\ bs\ ]\!]_{\overline{t \mapsto u}}$$
where $f\ (K\ \overline{x}\ \overline{t}) = bs$

$$\boxed{\mathrm{dom} : u \to h}$$

$$\begin{aligned}
&\mathrm{dom}\ K = 0 \\
&\quad \textbf{where}\ K : E \\
&\mathrm{dom}\ (K\ v\ \overline{u}) = \mathrm{dom}\ v + 1_\alpha \\
&\quad \textbf{where}\ K : E \to \alpha \to \overline{T} \to E
\end{aligned}$$

Figure 7.3: Binding specification evaluation

can consists of single variable or function calls. Hence, we need to define an interpretation of binding specifications and functions. This is a prerequisite for defining well-scopedness of terms and other boilerplate.

Figure 7.3 defines the interpretation $[\![\ bs\ ]\!]_{\vartheta}$ of a binding specification bs as a meta-level evaluation. Interpretation is always performed in the context of a particular constructor K. This is taken into account in the interpretation function: the parameter ϑ represents a *constructor local value environment* which assigns values to meta-variables that are used in K, i.e. it maps field labels for sub-terms to concrete de Brujin terms and global variables to concrete de Bruijn indices. We implicitly assume that all terms in ϑ are well-sortedness. In this section, we only use the subterm part of value environments. The global variable part is needed in the semantics of expressions and relations.

The result of the evaluation is a heterogeneous number h. Like in Section 5.2.2, we use 1_α as abbreviation for $(S_\alpha\ 0)$ and make use of addition. In case the binding specification item is a single-variable binding, the result is a one with the correct tag. In the interesting case of a function call $(f\ t_i)$, the evaluation pattern matches on the corresponding subterm $(\vartheta\ t_i)$ and interprets

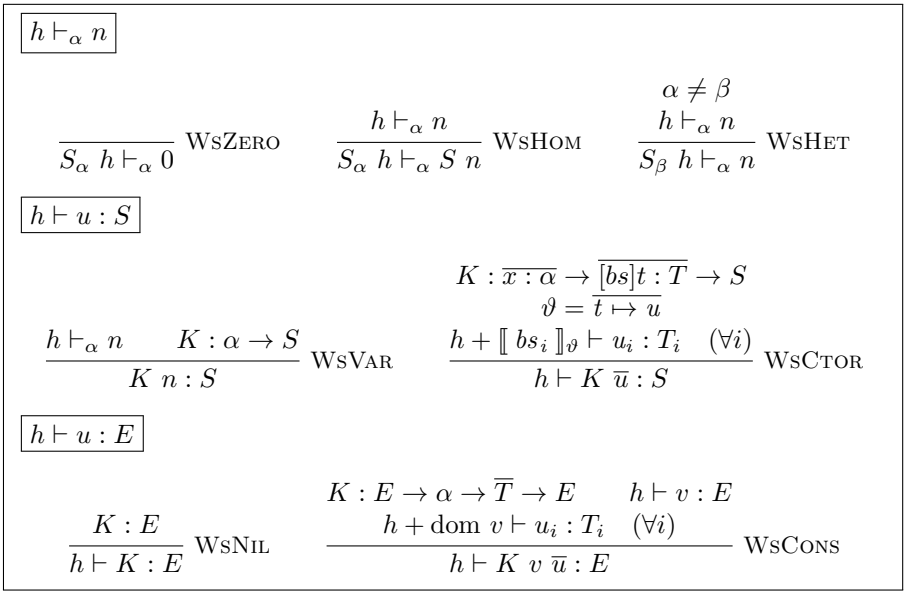


Figure 7.4: Well-scopedness of terms

the right-hand side of the appropriate function clause with respect to the new subterms. Note that we have ruled out function definitions for variable constructors. Thus, we do not need to handle that case here. The evaluation is well-defined since function declarations are required to be exhaustive and we only evaluate functions on well-sorted terms.

7.1.3 Well-scopedness

We now come to the definition of well-scoping for de Brujin terms. Figure 7.4 defines *well-scopedness relations* for de Bruijn indices, sort terms, and environment terms. The relation $h \vdash_\alpha n$ denotes that n is a well-scoped de Bruijn index for namespace α with respect to the variables in h. Rule WsHom which strips away one successor in the homogeneous case and rule WsHet that simply skips successors in the heterogeneous case. Rule WsZero forms the base case for $n = 0$ which requires that h has a successor tagged with α. This is a straightforward generalization of the well-scopedness relations that we defined for the $F_{\exists,\times}$ calculus in Section 5.8.

Rule WsVar delegates well-scopedness of variable constructors to the well-

scopedness of the index in the appropriate namespace. In rule WsCtor, the heterogeneous variable list h is extended for each subterm u_i with the result of evaluating its binding specification bs_i.

The relation $h \vdash u : E$ defines the well-scopedness of environment terms with respect to previously existing variables h. We will also write $\vdash u : E$ as short-hand for $0 \vdash u : E$. Note in particular that rule WsCons extends h with the dom of the prefix when checking the well-scopedness of associated data.

7.2 Expression Semantics

The semantics of expressions is an evaluation: given an assignment ϑ to all appearing meta-variables, we can evaluate an expression *sym* to a de Bruijn term u. Expressions can contain weakenings and substitutions and we have to give a proper interpretation for them in terms of the de Bruijn representation. These are boilerplate definitions that we define generically in Sections 7.2.1 and 7.2.2 before coming back to evaluation in Section 7.2.3.

7.2.1 Shifting and Weakening

Shifting adapts indices when a variable x is inserted into the context which is generically defined in Figure 7.5. The *shift* function is parameterized over the namespace α of variable x in which the shift is performed and the sort S of the term that the function operates on. In case of a variable constructor $K : \alpha \to S$, the index is shifted using the $shift_{\alpha,\mathbb{N}}$ function, which implements shifting indices for namespace α. For variable constructors of other namespaces, we keep the index unchanged. In the case of a regular constructor, we need to calculate the cut-offs for the recursive calls. This is done by evaluating the binding specification *bs* and weakening the cut-off c accordingly.

To avoid clutter, the definition presented here does not take subordination into account, i.e. for $F_{\exists,\times}$ a shifting of term variables inside a term will also recurse into types. Since types do not contain term variables this is effectively the identity function. This can of course be optimized. The paper [Keuchel et al., 2016] contains this optimized version.

Weakening Weakening is the transportation of a sort term to a bigger context where variables are only added at the end. Figure 7.5 shows the implementation of $weaken_S$ that iterates the 1-place $shift_{\alpha,S}$ function. Its second parameter h represents the domain of the context extension.

$$\boxed{\mathit{shift}_{\alpha,\mathbb{N}} : c \to n \to n}$$

$$\begin{aligned}
&\mathit{shift}_{\alpha,\mathbb{N}}\ 0\ n &&= S\ n\\
&\mathit{shift}_{\alpha,\mathbb{N}}\ (S_\alpha\ c)\ 0 &&= 0\\
&\mathit{shift}_{\alpha,\mathbb{N}}\ (S_\alpha\ c)\ (S\ n) &&= S\ (\mathit{shift}_{\alpha,\mathbb{N}}\ c\ n)\\
&\mathit{shift}_{\alpha,\mathbb{N}}\ (S_\beta\ c)\ n &&= \mathit{shift}_{\alpha,\mathbb{N}}\ c\ n\\
&\quad \textbf{where}\ \alpha \neq \beta
\end{aligned}$$

$$\boxed{\mathit{shift}_{\alpha,S} : c \to u \to u}$$

$$\begin{aligned}
&\mathit{shift}_{\alpha,S}\ c\ (K\ n) = \textbf{if}\ K : \alpha \to S\ \textbf{then}\ K\ (\mathit{shift}_{\alpha,\mathbb{N}}\ c\ n)\ \textbf{else}\ K\ n\\
&\mathit{shift}_{\alpha,S}\ c\ (K\ \overline{u}) = K\ \overline{\mathit{shift}_{\alpha,T}\ (c + [\![\ bs\ ]\!]_\vartheta)\ u}\\
&\quad \textbf{where}\\
&\qquad K : \overline{b : \beta} \to (\overline{[\,bs\,]\ t : T}) \to S\\
&\qquad \vartheta = \overline{t \mapsto u}
\end{aligned}$$

$$\boxed{\mathit{weaken}_S : u \to h \to u}$$

$$\begin{aligned}
&\mathit{weaken}_S\ u\ 0 &&= u\\
&\mathit{weaken}_S\ u\ (S_\alpha\ h) &&= \mathit{shift}_{\alpha,S}\ 0\ (\mathit{weaken}_S\ u\ h)
\end{aligned}$$

Figure 7.5: Shifting of terms

7.2.2 Substitution

Figure 7.6 also contains the definition of substitution. Like for shift, we define substitution by two functions. The function $\mathit{subst}_{\alpha,\mathbb{N}}\ c\ v\ n$ defines the operation for namespace α on indices by recursing on c and case distinction on n. If the index and the cut-off match, then the result is the term v. If the index n is strictly smaller or strictly larger than the cut-off c, then $\mathit{subst}_{\alpha,\mathbb{N}}$ constructs a term using the variable constructor for α. In the recursive cases, $\mathit{subst}_{\alpha,\mathbb{N}}$ performs the necessary weakenings when coming out of the recursion in the same order in which the binders have been crossed. This avoids a multiplace *weaken* on terms. The substitution $\mathit{subst}_{\alpha,S}$ traverses terms to the variable positions and weakens the trace according to the binding specification. As previously discussed v remains unchanged.

Like for shifting we can use subordination information to avoid recursing into sub-terms for which we statically know that they do not contain variables. The paper [Keuchel et al., 2016] also contains this optimized version.

$$
\begin{array}{l}
\boxed{subst_{\alpha,\mathbb{N}} : c \to v \to n \to u} \\
\quad subst_{\alpha,\mathbb{N}}\ 0\ \ \ \ \ v\ 0 \ \ \ \ \ = v \\
\quad subst_{\alpha,\mathbb{N}}\ 0\ \ \ \ \ v\ (S\ n) = K\ n \\
\quad\quad \textbf{where}\ K : \alpha \to T \\
\quad subst_{\alpha,\mathbb{N}}\ (S_\alpha\ x)\ v\ 0 \ \ \ \ \ = K\ 0 \\
\quad\quad \textbf{where}\ K : \alpha \to T \\
\quad subst_{\alpha,\mathbb{N}}\ (S_\alpha\ c)\ v\ (S\ n) = weaken_S\ (subst_{\alpha,\mathbb{N}}\ c\ v\ n)\ 1_\alpha \\
\quad subst_{\alpha,\mathbb{N}}\ (S_\beta\ c)\ v\ n \ \ \ \ \ = weaken_S\ (subst_{\alpha,\mathbb{N}}\ c\ v\ n)\ 1_\beta \\
\boxed{subst_{\alpha,S} : c \to v \to u \to u} \\
\quad subst_{\alpha,S}\ c\ v\ (K\ n) = \textbf{if}\ K : \alpha \to S\ \textbf{then}\ subst_{\alpha,\mathbb{N}}\ c\ v\ n\ \textbf{else}\ K\ n \\
\quad subst_{\alpha,S}\ c\ v\ (K\ \overline{u}) = K\ \overline{subst_{\alpha,T}\ (c + [\![\ bs\ ]\!]_\vartheta)\ v\ u} \\
\quad\quad \textbf{where} \\
\quad\quad\quad K : \overline{b : \beta} \to (\overline{[\,bs\,]\ t : T}) \to S \\
\quad\quad\quad \vartheta = \overline{t \mapsto u}
\end{array}
$$

Figure 7.6: Substitution of terms

7.2.3 Evaluation

We now define the semantics of symbolic expressions as an evaluation to concrete de Bruijn terms. Figure 7.7 contains the definition. The evaluation function takes as inputs a symbolic expression *sym*, the local scope *bs* of *sym* and a value environment ϑ for global and sort meta-variables.

Sort variables t are looked up in ϑ. We assume that these terms are in the same scope as t and hence they do not need to be adjusted. Global reference variables g are also looked up in ϑ but need to be adjusted by weakening with the interpretation of the current local scope *bs*. For variable constructors with locally bound variables b we determine the index by interpreting the difference bs' of the current scope and the scope where b was introduced. For regular constructors we recursively evaluate each expression for the sort fields in the local scope respectively extended with the symbolically evaluated binding specifications of the fields. Bindings $\overline{b}$ of regular constructors are dropped.

Symbolic syntactic operations are replaced by applications of the concrete versions. In case of symbolic weakening we need to evaluate the expression ar-

$$\boxed{eval_S : bs \to sym \to \vartheta \to u}$$

$$\begin{array}{lll}
eval_S\ bs & t & \vartheta = \vartheta\ t \\
eval_S\ bs & (K\ g) & \vartheta = K\ (\vartheta\ g + [\![\ bs\ ]\!]_\vartheta) \\
eval_S\ (bs, b, bs')\ (K\ b) & & \vartheta = K\ (0 + [\![\ bs'\ ]\!]_\vartheta) \\
eval_S\ bs & (K\ \overline{b}\ \overline{sym}) & \vartheta = \\
\multicolumn{3}{l}{\quad \overline{K\ eval_T\ (bs, \{\overline{b' \mapsto b}\}\ bs')\ sym\ \vartheta}} \\
\multicolumn{3}{l}{\quad \textbf{where}\ K : \overline{(b' : \alpha)} \to \overline{([bs']\ s : S)} \to T} \\
eval_S\ (bs, bs') & (weaken\ sym\ bs') & \vartheta = \\
\multicolumn{3}{l}{\quad weaken_S\ (eval_S\ bs\ sym\ \vartheta)\ [\![\ bs'\ ]\!]_\vartheta} \\
eval_S\ bs & (subst\ b\ sym_1\ sym_2)\ \vartheta = & \\
\multicolumn{3}{l}{\quad subst_{\alpha,S}\ 0\ (eval_T\ bs\ sym_1\ \vartheta)\ (eval_S\ (bs, b)\ sym_2\ \vartheta)} \\
\multicolumn{3}{l}{\quad \textbf{where}\ K : \alpha \to T}
\end{array}$$

Figure 7.7: Expression evaluation

gument in the smaller scope bs and weaken it with the interpretation of bs' for which we use the concrete multi-place weakening $weaken_S$. We restricted symbolic substitution to only allow substituting the last introduced variable and hence always substitute the index 0. The expression arguments of substitution need to be evaluated in their respective local scopes.

7.3 Relation Semantics

In this section we define the semantics of relations. A prerequisite is interpretation of environment lookups. This the paramount boilerplate operation on environments that we define generically in Section 7.3.1. Premises of rules can contain rule binding specifications which we need to interpret. They define the environment extensions for the corresponding premise. In Section 7.3.2 we show how to evaluate rule binding specifications to concrete environment terms that represent the extensions. Finally, Section 7.3.3 discusses the interpretation of relations as derivation trees.

$$\boxed{(n : \overline{u}) \ \in_{\alpha,E} \ v}$$

$$\frac{\text{dom } v \vdash u_i : S \quad (\forall i) \qquad K : E \to \alpha \to \overline{S} \to E}{(0 : \overline{\mathit{weaken}_S \ u \ 1_\alpha}) \ \in_{\alpha,E} \ (K \ v \ \overline{u})} \ \textsc{InHere}$$

$$\frac{(n : \overline{u}) \ \in_{\alpha,E} \ v \qquad K : E \to \alpha \to \overline{S} \to E \qquad K' : E \to \beta \to \overline{T} \to E}{(\mathit{weaken}_\alpha \ n \ 1_\beta : \overline{\mathit{weaken}_S \ u \ 1_\beta}) \ \in_{\alpha,E} \ (K' \ v \ \overline{u'})} \ \textsc{InThere}$$

Figure 7.8: Environment lookup

7.3.1 Environment lookups

Lookup is a partial function. For that reason, we define it as a relation $(n : \overline{u}) \ \in_{\alpha,E} \ V$ that witnesses that looking up the index n of namespace α in the environment term V is valid and that $\overline{u}$ is the associated data. Figure 7.8 contains the definition.

Rule InHere forms the base case where $n = 0$. In this case the environment term needs to be a cons for namespace α. Note that well-scopedness of the associated data is included as a premise. This reflects common practice of annotating variable cases with with well-scopedness conditions. By moving it into the lookup itself, we free the user from dealing with this obligation explicitly. We need to *weaken* the result of the lookup to account for the binding.

Rule InThere encodes the case that the lookup is not the last cons of the environment. The rule handles both the homogeneous $\alpha = \beta$ and the heterogeneous case $\alpha \neq \beta$ by means of weakening the index n. The associated data is also weakened to account for the new β binding.

7.3.2 Rule Binding Specifications

Similar to evaluation of binding specifications of sorts, we define the semantics of rule binding specifications of relations as an evaluation. The result in this case are environment terms instead of domains.

In case of a new single variable binding $b \to \overline{\mathit{sym}}$ with a binding variable b of namespace α we construct the environment term using the constructor K of E, the result $(\mathit{eval}_{E,\mathcal{L},\vartheta} \ \mathit{rbs})$ of recursively evaluating the prefix rule binding specification, and the evaluated associated data. The associated data are expressions which need to be evaluated also. We get the local scope bs for the

$$\boxed{eval_{E,\mathcal{L},\vartheta} : rbs \to u}$$

$$\begin{aligned}
&eval_{E,\mathcal{L},\vartheta}\ \epsilon && = 0\\
&eval_{E,\mathcal{L},\vartheta}\ (rbs, b \to \overline{sym}) && = K\ (eval_{E,\mathcal{L},\vartheta}\ rbs)\ (\overline{eval_S\ bs\ sym\ \vartheta})\\
&\quad \textbf{where}\ (b : \alpha) \in \mathcal{L}\\
&\qquad\qquad K : E \to \alpha \to \overline{T} \to E\\
&\qquad\qquad \mathcal{L} \vdash_E rbs \Downarrow bs\\
&eval_{E,\mathcal{L},\vartheta}\ (rbs, f\ j) && = append_E\ (eval_{E,\mathcal{L},\vartheta}\ rbs)\ (\vartheta\ (f, j))
\end{aligned}$$

Figure 7.9: Evaluation of rule binding specifications

evaluation of expressions by flattening the prefix of the rule binding specification. For a function call on a judgement j, we look up the corresponding environment term in the local value environment ϑ and append it to the result of recursively evaluating the prefix.

7.3.3 Derivations

Semantics of relations are derivation trees of judgements. However, for the purpose of deriving boilerplate lemmas the interesting structure lies primarily in the symbolic expressions used to define the rules of the relations. For example, none of the elaborations for boilerplate lemmas make use of representations of derivations. The lemmas need to induct over derivations but we leave this aspect to the concrete implementation. Therefore, we only introduce declaration heads as notation for use in subsequent sections.

Relations are generically modelled by judgements of the form

$$\boxed{u_E? \models_R \overline{u_t}; \overline{u_f}}$$

where u_E is an optional environment term and $\overline{u_t}$ are the sort indices of the relations. Rather than interpreting binding functions as computations over derivation trees, we include the results as indices $\overline{u_f}$ of the judgements. To further simplify the presentation in the subsequent chapters, we assume that all relations have an environment E and ignore outputs of functions, i.e. we only consider judgements of the form

$$\boxed{u_E \models_R \overline{u_t}}.$$

Chapter 8

Elaboration

For the meta-theoretical formalization of programming language we need properties of the semantical definitions. For instance, for proofs of type-safety we generally need substitution properties of the typing relation, among other boilerplate properties.

There are multiple ways to prove the boilerplate lemmas for use in mechanizations. We can develop a *code generator* that produces code for a proof-assistant. Alternatively, we can develop a *datatype-generic* library inside a proof-assistant. Both approaches have different trade-offs.

Code Generators A code generator can parse textual Knot specifications and generate definitions and proofs for use in a proof-assistant.

In terms of usability, this approach has clear advantages. The generated definitions of datatypes and functions can be close to what a human prover would write manually. Hence, the user can inspect the code to better understand the provided functionality. Moreover, since the code generator can specialize the statements of lemmas to the given specification, it is much easier for the user to look at the provided proofs and for proof automation to apply the provided proofs.

The main downside of a code generator is that we do not know whether the correct code is generated in all cases. As a consequence, the proof assistant still has to check the generated code.

Coq enables a particular staged approach to code generation that leverages its proof automation facilities. The idea is that the code generator does not

directly generate custom proof terms, but instead generates an invocation of a Coq proof script. When this proof script is run, it builds the actual custom proof term. This reduces the development effort, because the generator does not have to implement its own proof generation. Moreover, the approach is more flexible because we can change the definitions of the scripts without changing the generator. However, generic proof scripts tend to be brittle and can take a long time to execute. Furthermore, reasoning about the correctness of the code generator not only involves the language of the proof assistant, but also that of the scripting system.

Generic Library Alternatively, we can develop a datatype-generic library with (well-formed) Knot specifications as the codes of a generic universe. The main benefit of such a library is, that its code, including all proofs, can be checked once and for all, and independently of a concrete specification. Hence, instantiating the library is guaranteed to yield valid theorems for all specifications. Moreover, it shows that Knot embodies all necessary information and constraints to derive the boilerplate.

There are several downsides to this approach. For usability, it is important to hide the internals of the library and its complexity, since the user should not have expert knowledge about datatype-generic programming to use the library. Yet, there are several ways in which the implementation and its complexity may be exposed. Firstly, the simplification of boilerplate functions may reveal the generic implementation. Secondly, the provided induction principle over generically defined datatypes is (usually) not the one the user would expect. The above problems are usually mitigated in datatype-generic programming by working with user-defined types instead, and by writing isomorphisms to the generic representation. This means however, that some definitions like well-scopedness predicates, which are clearly boilerplate, have to be user-defined as well, which defeats the purpose of a generic solution to this kind of boilerplate. Luckily, it is possible to make the interface of datatype-generic libraries aware of isomorphisms so that the user is spared from applying the isomorphisms manually.

Automation of proofs may also be impaired when using lemmas of a generic library in comparison to manually proved or generated lemmas. For instance, the statement of a generic lemma may not unify with all goals it applies to before it has been properly specialized to a given specification. This impacts automatic proof search, in particular for user-defined relations, which can have arbitrary arities. Related to that is the problem of proof discovery. If the user wants to look at the lemmas a library provides, he sees only the generic lemmas

instead of the ones specialized for his specification. Conceptually, this should not pose a problem since in theory a proof assistant could specialize the types of lemmas when instantiating the generic code for a specification. However, currently none of the available proof assistants provide such functionality.

Finally, developing a datatype-generic library for KNOT is technically highly challenging. Constraints like well-formedness of binding specifications and expressions needs to be part of the universe code. Looking up meta-variables can potentially introduce partiality which needs to be dealt with. Furthermore, there is little research on writing datatype-generic libraries at this scale: not only is datatype-generic programming more established than datatype-generic proving, but also, research on generic proofs usually deals with universes of algebraic datatypes and does not extend to user-defined relations which are part of KNOT specifications.

Hybrid Approaches The main trade-off of the above two approaches is between the confidence in the correctness and the usability. Ideally, we could combine both approaches in order to get the best of both worlds, and to mitigate the respective disadvantages.

The obvious hybrid approach is to develop a generic library and address the usability problems by developing a user-friendly code generator around it. The code generator can instantiate the library with a user-defined specification and specialize the generic definitions and lemmas as much as possible. However, the technical challenges to develop a generic library remain. To ensure that all generic definitions are hidden, we need to generate specialized definitions for every generic definition that appears in the user facing interface. This ranges from obvious user-defined datatype declarations for sorts, environments and relations to generically derived inductive definitions like domains, well-scopedness relations and environment look-ups. However, the specialization adds duplication between the code generator and the generic implementation.

An alternative hybrid approach, is to use a code-generator as the principle implementation and to transfer as much confidence in correctness as possible from the generic approach. This is the path that we take and the remainder of this chapter outlines our methodology to do so. To bolster our confidence in code generation we have first built LOOM, a datatype-generic library in COQ, and then transferred its proof term generation to NEEDLE, a code generator in Haskell that produces COQ code.

Methodology LOOM defines de Bruijn interpretation of KNOT specification generically and also implemented boilerplate functions and lemmas generically.

Elaborating datatype and function definitions in NEEDLE is comparably easy. The main obstacle, that we need to solve, is how to turn a generic proof of LOOM into an elaboration function in NEEDLE. One can aim to implement an elaboration that follows the *same strategy as the generic proof*. However, the question remains if this is a faithful implementation of the same proof and if it indeed is correct in all cases, especially when proof scripts are used.

We solve this problem by factoring the generic proofs of LOOM into elaboration proofs. The first part are elaborations that produce for any given specification specialized proofs and the second part are formal verifications of the correctness of the elaborations.

Since the internals of the proofs assistant are not directly accessible for elaboration or formal verification, and the language of the proof assistant is too big, we choose a different target for the elaboration: namely intermediate domain-specific witness languages. For each class of lemmas we develop a new witness language specific for that class.

After developing the elaboration functions, their formal verification can be divided into three different parts, which together represent alternatives to the generic proofs:

1. A formal semantics of the witness language.
2. The correctness of the elaboration, i.e. the produced witness indeed encodes a proof of the desired lemma w.r.t. the chosen semantics.
3. A soundness proof of the semantics of the witness language, e.g. the statement that a witness proves w.r.t. to the chosen semantics is valid (in our meta-language).

Subsequently, we can implement the same elaborations in the code generator NEEDLE. However, this does not establish formal correctness of NEEDLE. The confidence in the correctness of NEEDLE still relies on some factors.

1. *The elaboration functions from* KNOT *specifications to the witness languages is correctly ported from* COQ *to* HASKELL. The elaborations are recursive and pure functions[1] over algebraic dataypes which both COQ and HASKELL support. Therefore, we port the elaboration code by literal translation from COQ to HASKELL. In fact, most HASKELL elaboration functions were derived by a copy & paste of the COQ code.
2. *The elaboration of the witness language into the language of the target proof-assistant is correctly implemented.* For our witness languages, this

[1] In reality, the elaboration functions are exclusively folds.

translation is straightforward, it is compositional and each witness primitive is either implemented by an application of a lemma or a rule.

3. The formal elaborations target only very specific parts of the boilerplate lemmas. For instance, our formal proof elaborations exclusively deal with inductive steps of induction proofs. Setting up the induction itself is done in unchecked *glue code*. However, in comparison this glue code is less fragile.

Overview There are too many lemmas to give an exhaustive account of all the necessary elaborations. Instead we concentrate on representative and key lemmas.

The first sections in this chapter show how the witness language methodology is applied to several classes of boilerplate lemmas. Section 8.1.1 discusses common interaction lemmas between syntactic operations. Section 8.2 deals with the well-scopedness lemmas that we discussed in Section 5.2.4 and 5.2.5, i.e. lemmas that prove that indices of relations are always well-scoped. Section 8.3 covers shifting and substitution lemmas for relations. Each of the Sections looks at the class of lemmas that is being discussed, the domain-specific witness language for that class, an elaboration for a representative lemma, and the verification of the correctness of the elaboration.

Section 8.4 discusses the generic library implementation Loom of Knot in Coq, and Section 8.5 covers the implementation of our code generator Needle. We look at related work in Section 8.6 and finally conclude in Section 8.7 with our contributions.

8.1 Interaction Lemmas

Formalizations involve a number of interaction boilerplate lemmas between *shift*, *weaken* and *subst*. These lemmas are for example needed in weakening and substitution lemmas for typing relations.

In this Section, we develop an syntax-directed elaboration for these lemmas. Since this class of lemmas state the equality of different applications of operations, we develop a witness language for term equality of our de Bruijn representation.

We discuss the two types of interaction lemmas in Section 8.1.1 and develop a semi-formal induction proof for one of them in Section 8.1.2. The formal witness language is developed in Section 8.1.3. Finally, the elaboration is presented in Section 8.1.4.

8.1.1 Overview

There are two distinct types of interaction lemmas: commutation lemmas and a cancelation lemma.

Commutation

Two operation always commute when they are operating on different variables. For instance, weakening of terms by introducing two distinct variables x and y

$$\Gamma, \Delta_1, \Delta_2 \rightsquigarrow \Gamma, x, \Delta_1, y, \Delta_2$$

can be implemented by 2 consecutive shifts. The order of these shifts is irrelevant, which we have to prove, i.e. we have the following commuting diagram:

$$\begin{array}{ccc} \Gamma, \Delta_1, \Delta_2 & \xrightarrow{shift_\beta\ h_2} & \Gamma, \Delta_1, y, \Delta_2 \\ \Big\downarrow{\scriptstyle shift_\alpha\ (h_1+h_2)} & & \Big\downarrow{\scriptstyle shift_\alpha\ (h_1+1_\beta+h_2)} \\ \Gamma, x, \Delta_1, \Delta_2 & \xrightarrow[shift_\beta\ h_2]{} & \Gamma, x, \Delta_1, y, \Delta_2 \end{array}$$

where $h_1 := \text{dom}\ \Delta_1$ and $h_2 := \text{dom}\ \Delta_2$, α is the namespace of x, and β the namespace of y. Usually only the special case $\Delta_2 \equiv \epsilon$ of this lemma is used. However, for the induction to go through the more general case needs to be proved. Also the lemma can be homogenous, i.e. $\alpha = \beta$, or heterogeneous, i.e. $\alpha \neq \beta$.

Lemma 9.

$$\forall u, h_1, h_2. shift_\alpha\ (h_1 + 1_\beta + h_2)\ (shift_\beta\ h_2\ u) = shift_\beta\ h_2\ (shift_\alpha\ (h_1 + h_2)\ u)$$

Similar lemmas hold for the commutation of a substitution and a shifting, and two substitutions. Extra care needs to be taken when a substitution is involved, since the substitute(s) may also need to be changed.

Cancelation

When operating on the same variable, an introduction of a variable with a shift and cancels a subsequent substitution:

$$\begin{array}{ccc} \Gamma, \Delta & \xrightarrow{shift_\alpha\ h} & \Gamma, x, \Delta \\ & {\scriptstyle id}\searrow & \Big\downarrow{\scriptstyle subst_\alpha\ h\ v} \\ & & \Gamma, \Delta \end{array}$$

where $h := \text{dom}\ \Delta$.

Lemma 10.

$$\forall v, u, h. subst_\alpha\ h\ v\ (shift_\alpha\ h\ u) = u$$

8.1.2 Semi-formal Shift Commutation

In this section we present a proof of the shift commutation lemma 9 that discusses all necessary proof steps in detail, but that is not completely formal. The proof of the lemma proceeds by induction over the structure of the term u. The cut-offs of recursive calls are calculated using the defined binding specification, but for the outer shifts of the equation, the cut-off calculations are performed on shifted terms and thus do not match the form of the induction hypotheses. We first need to prove an auxiliary property of KNOT's de Brujin interpretation, namely that evaluation of binding functions and binding specifications remains stable under shifting and substitution.

Stability of Binding Specifications

This property can be seen as an enforcement of lexical scoping: syntactic operations on (free) variables do not change the scoping structure of bound variables. The stability is enforced in the design of KNOT: by ruling out functions over sorts with variables, function evaluation can never reach variable cases and thus their results only depends on the parts of the structure that are left unchanged by syntactic operations.

The following lemma states the stability property for function evaluation.

Lemma 11. *For all terms t of sort S and all $f : S \to \overline{\alpha}$ the following holds:*

1. $\forall \beta, c. [\![f]\!] (shift_\beta\ c\ t) = [\![f]\!] (t)$
2. $\forall \beta, s, c. [\![f]\!] (subst_\beta\ c\ s\ t) = [\![f]\!] (t)$

Proof. By induction over the structure of t. The variable case is ruled out for the sort S, so we only need to consider the regular cases. For each regular constructor K of S the goal follows by nested induction of the right-hand side of f for K. □

The nested induction of the lemma above deserves essentially proves that evaluation of binding specifications is invariant under shifting and substitution. This is a useful result of its own.

Corollary 12. *Let bs be a binding specification and* $\vartheta = \overline{t_i \mapsto u_i}^i$ *a local value environment. For given cut-offs* $\overline{h_i}^i$ *and substitutes* $\overline{v_i}^i$ *define the following value environments*

$$\begin{array}{lcl} \vartheta'(t_i) & := & \mathit{shift}_\alpha\ h_i\ u_i \\ \vartheta''(t_i) & := & \mathit{subst}_\alpha\ h_i\ v_i\ u_i \end{array}$$

The following holds

1. $\forall bs.[\![\ bs\ ]\!]_{\vartheta'} = [\![\ bs\ ]\!]_{\vartheta}$,
2. $\forall bs.[\![\ bs\ ]\!]_{\vartheta''} = [\![\ bs\ ]\!]_{\vartheta}$.

Proof. By induction over *bs* and using Lemma 11 for function calls. □

Shift Commutation

We now come to the proof of the shift commutation lemma itself. A prerequisite is a proof of the lemma in the case of a variable. The variable case lemma is largely independent of a concrete KNOT specification, only the declared namespaces are involved, but not the structure of user-defined sorts. This allows a generic implementation of the variable case without the need of a special elaboration. We therefore assume that the variable case is given.

Proof of Lemma 9. The proof of the lemma proceeds by induction over u. As discussed the variable constructor is easy to handle. Hence, we focus on the the inductive steps of the regular constructors.

For the regular case suppose that we need to show the statement for $K\ \overline{u}$ with $K : \overline{x : \alpha} \to \overline{[bs]t : T} \to S$. Define the constructor local value environment ϑ for the inner shift, and two value environments ϑ' and ϑ'' for the outer shifts of the equation

$$\begin{array}{lcl} \vartheta(t_i) & := & u_i \\ \vartheta'(t_i) & := & \mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u_i \\ \vartheta''(t_i) & := & \mathit{shift}_\alpha\ ((h_1 + h_2) + [\![\ bs\ ]\!]_\vartheta)\ u_i \end{array}$$

The lemma follows by applying the following equational reasoning to each field u with binding specification bs:

$$
\begin{array}{rcl}
eqh & ::= & \mathit{refl} \mid \mathit{sym}\ eqh \mid \mathit{trans}\ eqh\ eqh \mid \mathit{cong}_{S_\alpha}\ eqh \mid \mathit{cong}_{+}\ eqh\ eqh \\
 & \mid & \mathit{assoc}_{+} \mid \mathit{shift}_\alpha\ f\ t \\
equ & ::= & \mathit{refl} \mid \mathit{sym}\ equ \mid \mathit{trans}\ equ\ equ \mid \mathit{ih} \\
 & \mid & \mathit{cong}_{\mathit{shift}_\alpha}\ eqh\ equ
\end{array}
$$

Figure 8.1: Equality Witness DSL

$$
\begin{array}{lll}
 & \mathit{shift}_\alpha\ (((h_1 + 1_\beta) + h_2) + [\![\ bs\ ]\!]_{\vartheta'}) & (\mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u) \\
\equiv & \text{By Lemma 12.} & \\
 & \mathit{shift}_\alpha\ (((h_1 + 1_\beta) + h_2) + [\![\ bs\ ]\!]_\vartheta) & (\mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u) \\
\equiv & \text{By associativity.} & \\
 & \mathit{shift}_\alpha\ ((h_1 + 1_\beta) + (h_2 + [\![\ bs\ ]\!]_\vartheta)) & (\mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u) \\
\equiv & \text{By the inductive hypothesis.} & \\
 & \mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta) & (\mathit{shift}_\alpha\ (h_1 + (h_2 + [\![\ bs\ ]\!]_\vartheta))\ u) \\
\equiv & \text{By associativity.} & \\
 & \mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta) & (\mathit{shift}_\alpha\ ((h_1 + h_2) + [\![\ bs\ ]\!]_\vartheta)\ u) \\
\equiv & \text{By Lemma 12.} & \\
 & \mathit{shift}_\beta\ (h_2 + [\![\ bs\ ]\!]_{\vartheta''}) & (\mathit{shift}_\alpha\ ((h_1 + h_2) + [\![\ bs\ ]\!]_\vartheta)\ u)
\end{array}
$$

□

8.1.3 Term Equality Witnesses

We now come to the first ingredient of a formal proof of the shift commutation lemma, a witness language that is the target of a formal elaboration. The elaboration we develop covers the equational reasoning of the induction steps of Lemma 9. Figure 8.1 shows a grammar for the witness language. There are two sorts: *eqh* which encodes equalities between domains and *equ* that encodes equalities between de Bruijn terms. Both sorts have productions for *refl*, *sym* and *trans* that represent the *reflexivity*, *symmetry* and *transitivity* of an *equivalence relation*.

The remaining productions encode additional equalities. The productions for cong_{S_α} respectively cong_{+} encode congruences for the successors and plus functions, and assoc_{+} witnesses the associativity of plus. The stability of evaluating a binding function f is added as the primitive $\mathit{shift}_\alpha\ f\ t$. For equality of terms we have congruence of shifting $\mathit{cong}_{\mathit{shift}_\alpha}$ as an additional equality, and *ih* that denotes the application of an induction hypothesis.

The interpretation of the domain equality witnesses is show in Figure 8.2. The judgement $\models_\vartheta eqh : h_1 \equiv h_2$ denotes that *eqh* witnesses the equality of h_1 and h_2 under the value environment ϑ. The equivalence properties, congruence rules and associativity are entirely standard. The interpretation of the shift

$\boxed{\models_\vartheta eqh : h \equiv h}$

$$\frac{}{\models_\vartheta refl : h \equiv h}\ \textsc{EqhRefl} \qquad \frac{\models_\vartheta eqh : h_1 \equiv h_2}{\models_\vartheta sym\ eqh : h_2 \equiv h_1}\ \textsc{EqhSym}$$

$$\frac{\models_\vartheta eqh_1 : h_1 \equiv h_2 \qquad \models_\vartheta eqh_2 : h_2 \equiv h_3}{\models_\vartheta trans\ eqh_1\ eqh_2 : h_1 \equiv h_3}\ \textsc{EqhTrans}$$

$$\frac{\models_\vartheta eqh : h_1 \equiv h_2}{\models_\vartheta cong_{S_\alpha}\ eqh : S_\alpha\ h_1 \equiv S_\alpha\ h_2}\ \textsc{EqhCongSucc}$$

$$\frac{\models_\vartheta eqh_1 : h_1 \equiv h_3 \qquad \models_\vartheta eqh_2 : h_2 \equiv h_4}{\models_\vartheta cong_+\ eqh_1\ eqh_2 : h_1 + h_2 \equiv h_3 + h_4}\ \textsc{EqhCongPlus}$$

$$\frac{}{\models_\vartheta assoc_+ : (h_1 + h_2) + h_3 \equiv h_1 + (h_2 + h_3)}\ \textsc{EqhAssocPlus}$$

$$\frac{}{\models_\vartheta shift_\alpha\ f\ t : [\![f]\!](shift_\alpha\ h\ (\vartheta\ (t))) \equiv [\![f]\!](\vartheta\ (t))}\ \textsc{EqhFunShift}$$

Figure 8.2: Interpretation of Domain Equality Witnesses

stability primitive $shift_\alpha\ f\ t$ is exactly the statement of the stability lemma 11.

The interpretation of the term equality witnesses is the relation $v_0 \equiv v_1 \models_\vartheta eqh : u_0 \equiv u_1$ defined in Figure 8.3 which denotes that eqh witnesses the equality of $u_0 \equiv u_1$ under the hypothesis $v_0 \equiv v_1$. The only interesting rule is EquIH, which interprets ih by using the equality of the hypothesis.

A property that we have to prove is that the semantics of the witness languages is soundness with respect to our meta-language.

Lemma 13 (Soundness)**.** *The domain and term equality witness languages are sound:*

$$\frac{\models_\vartheta eqh : h_1 \equiv h_2}{h_1 \equiv h_2} \qquad \frac{v_0 \equiv v_1 \models_\vartheta equ : u_1 \equiv u_2 \qquad v_0 \equiv v_1}{u_1 \equiv u_2}$$

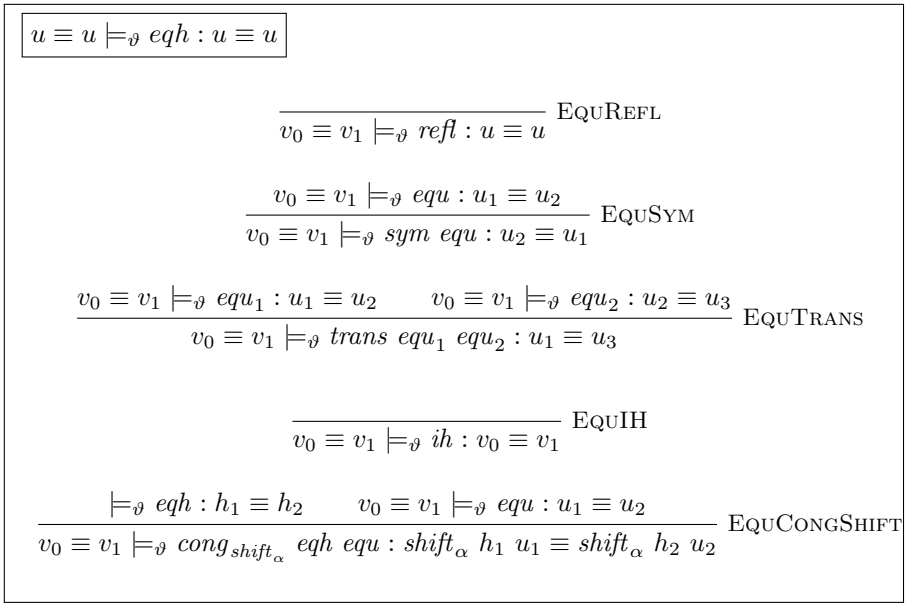


Figure 8.3: Interpretation of Term Equality Witnesses

8.1.4 Proof Elaboration

We can now turn into the formal elaboration into equality witnesses of the auxiliary shift-stability lemma and of the induction step of the shift commutation lemma.

Elaboration of Stability

The elaboration function in Figure 8.4 produces an equality witness for Corollary 12. The corollary is proven by induction over list-like binding specifications. The elaboration function follows the same recursive structure and is a fold over binding specifications. When the binding specification contains a function call, the stability axiom is used. Congruences are used to make sure we are in the proper position.

It remains to proof that the elaboration function indeed produces a witness for Corollary 12, which is done in the following lemma.

Lemma 14 (Correctness of Stability Elaboration)**.** *The elaboration for the stability of binding specification evaluation under shifting is correct.*

$$
\begin{array}{l}
stable_{shift_\alpha} : bs \to eqh \\
stable_{shift_\alpha}\ [\,] \quad\quad\ = refl \\
stable_{shift_\alpha}\ (bs, x) \ \ = cong_{S_\beta}\ (stable_{shift_\alpha}\ bs) \\
\quad \textbf{where}\ x : \beta \in \mathcal{L} \\
stable_{shift_\alpha}\ (bs, f\ t) = cong_{+}\ (stable_{shift_\alpha}\ bs)\ (shift_\alpha\ f\ t)
\end{array}
$$

Figure 8.4: Elaboration of Shift Stability

$$
\frac{eqh = stable_{shift_\alpha}\ bs \qquad \vartheta = \overline{u_i}^{\,i} \qquad \vartheta' = \overline{shift_\alpha\ h_i\ u_i}^{\,i}}{\models_\vartheta eqh : [\![\ bs\ ]\!]_{\vartheta'} \equiv [\![\ bs\ ]\!]_\vartheta}
$$

Elaboration of Shift Commutation

Figure 8.5 contains the elaboration function *comm* for the inductive step of the shift commutation lemma. We split the semi-formal proof of Section 8.1.2 into three parts

1. the reasoning steps before the application of the induction hypothesis, which are encoded by $comm_1$,
2. the application of the induction hypothesis, and
3. the remaining reasoning steps, which are encoded by $comm_2$.

The following lemma states that *comm* correctly elaborates the induction step of Lemma 9 the defined abbreviations $v0, v1$ are exactly the term of the equality given by the induction hypothesis in Lemma 9, and $u0, u1$ are the terms of the goal.

Lemma 15 (Correctness of Shift Commutation Elaboration)**.** *The elaboration for the shift commutation witness is correct.*

$$
\frac{\begin{array}{c}
equ = comm\ bs \qquad \vartheta = \overline{u_i}^{\,i} \\
v0 = shift_\alpha\ ((h_1 + 1_\beta) + (h_2 + [\![\ bs\ ]\!]_\vartheta))\ (shift_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u) \\
v_1 = shift_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ (shift_\alpha\ (h_1 + (h_2 + [\![\ bs\ ]\!]_\vartheta))\ u) \\
u0 = shift_\alpha\ (((h_1 + 1_\beta) + h_2) + [\![\ bs\ ]\!]_{\vartheta'})\ (shift_\beta\ (h_2 + [\![\ bs\ ]\!]_\vartheta)\ u) \\
u1 = shift_\beta\ (h_2 + [\![\ bs\ ]\!]_{\vartheta''})\ (shift_\alpha\ ((h_1 + h_2) + [\![\ bs\ ]\!]_\vartheta)\ u)
\end{array}}{v_0 \equiv v_1 \models_\vartheta equ : u_0 \equiv u_1}
$$

$$
\begin{array}{lll}
comm_1 & : & bs \to equ \\
comm_1\ bs & = & cong_{shift_\alpha} \\
 & & \quad (trans\ (cong_+\ refl\ (stable_{shift_\beta}\ bs))\ assoc_+) \\
 & & \quad refl \\
comm_2 & : & bs \to equ \\
comm_2\ bs & = & cong_{shift_\beta} \\
 & & \quad (cong_+\ refl\ (stable_{shift_\alpha}\ bs)) \\
 & & \quad (cong_{shift_\alpha}\ assoc_+\ refl) \\
comm & : & bs \to equ \\
comm\ bs & = & trans\ (comm_1\ bs)\ (trans\ ih\ (sym\ (comm_2\ bs)))
\end{array}
$$

Figure 8.5: Elaboration of Shift Commutation

8.2 Well-Scopedness

In this section, we develop a proof elaboration of well-scopedness lemmas for user defined relations. Similar to the previous section, we begin by giving a semi-formal overview of the proof steps before developing a formal elaboration.

The well-scopedness lemmas state that indices of relations with environments are well-scoped in the domain of the environment. For example for the formalized typing relation $\Gamma \vdash_{\text{tm}} e : \tau$ of $F_{\exists,\times}$ from the Overview Section 5.2 we want to prove the following two well-scopedness lemmas

$$\frac{\vdash E \qquad E \vdash_{\text{tm}} t : T}{E \vdash_{\text{tm}} t} \qquad \frac{\vdash E \qquad E \vdash_{\text{tm}} t : T}{E \vdash_{\text{ty}} T}$$

Both lemmas are proved by induction of the typing derivation and the induction steps basically boil down to proving that the expressions in the conclusion of a rule are well-scoped assuming that the expressions in the premises are well-scoped. Consider the typing rule for type application of $F_{\exists,\times}$

$$\frac{E \vdash_{\text{tm}} t : \forall \bullet . T \qquad E \vdash_{\text{ty}} S}{E \vdash_{\text{tm}} (e!S) : (\text{subst}_{\text{ty}}\ 0\ S\ T)}\ \text{TTApp}$$

To prove that the type $(\text{subst}_{\text{ty}}\ 0\ S\ T)$ of the conclusion is well-scoped we have to use the substitution lemma for well-scopedness of types. This still leaves us with the obligation to prove that the types represented by the

$$
\begin{array}{lll}
wn & ::= & hyp\ g\ \alpha \mid local\ \alpha\ bs \mid weaken\ wn\ bs \mid strengthen\ wn\ bs \mid varinv\ K\ ws \\
ws & ::= & hyp\ bs\ sym \mid var\ K\ wn \mid reg\ K\ \overline{ws} \mid reginv\ K\ n\ ws \\
 & \mid & weaken\ ws\ bs \mid strengthen\ ws\ bs \mid subst\ ws_1\ ws_2 \\
H & ::= & \overline{(g : \alpha)}, \overline{([bs]sym : S)}
\end{array}
$$

Figure 8.6: Well-Scopedness Witness DSL

meta-variables S and T are well-scoped. The well-scopedness of S is given by the second premise. The induction hypothesis for the first premise gives us the well-scopedness of $(\forall \bullet .T)$. By inversion of the well-scopedness rule for universal quantification we can conclude that T is well-scoped.

The next two sections develop the formal elaboration of the inductive steps. Section 8.2.1 presents the domain-specific witness language and Section 8.2.2 presents the elaboration function from symbolic expressions to witnesses.

8.2.1 Witnesses of Well-Scoping

Figure 8.6 defines the grammar of the witnesses for indices *wn* and for terms of sorts *ws* which are mutually recursive. The interpretation witnesses is relative to a fixed rule local environment $\mathcal{L} = \overline{(r@\alpha)}, \overline{([bs_b]b : \beta)}, \overline{[bs_t]t : T}$ and also a fixed value environment $\vartheta = \overline{(r \mapsto n)}, \overline{(t \mapsto u)}$. Furthermore, we use an environment H to hold hypotheses, which is populated from both the induction hypotheses of the rule and from well-scopedness premises. The witnesses describe the use of a hypothesis *hyp*, the well-scopedness of a local reference *local*, the application of a constructor rule *var* or *reg*, the inversion of a constructor rule *varinv* or *reginv*, use of a weakening lemma *weaken* or of a strengthening lemma *strengthen* (the inversion of weakening), or use of a substitution lemma *subst*.

Figure 8.7 contains selected rules that define the intended meaning of the proof terms with respect to the de Bruijn representation. (See Appendix A.2 for the remainder.) The relation $(wn) \models_\vartheta (h \vdash_n \alpha)$ denotes that wn witnesses that n is a well-scoped de Bruijn index for namespace α with respect to h and $(ws) \models_\vartheta (h \vdash_S u)$ denotes that ws witnesses that u is a well-scoped in the de Bruijn term of sort S with respect to h. These two relations are mutually-recursive and completely syntax directed in wn respectively ws.

Also note, that both relations are parameterized by H, ϑ and h_0, an additional parameter that represents the outer scope, i.e. the domain of the implicit environment during the proof of the well-scopedness lemma. The binding

$\boxed{(wn) \models_\vartheta (h \vdash_\alpha n)}$

$$\frac{(g : \alpha) \in H}{(hyp\ g\ \alpha) \models_\vartheta (h_0 \vdash_\alpha \vartheta\ g)}\ \textsc{WnHyp}$$

$$\frac{K : \alpha \to S \qquad (ws) \models_\vartheta (h \vdash_S K\ n)}{(varinv\ K\ ws) \models_\vartheta (h \vdash_\alpha n)}\ \textsc{WnVarInv}$$

$\boxed{(ws) \models_\vartheta (h \vdash_S u)}$

$$\frac{K : \alpha \to S \qquad (wn) \models_\vartheta (h \vdash_\alpha n)}{(var\ K\ wn) \models_\vartheta (h \vdash_S K\ n)}\ \textsc{WsVar}$$

$$\frac{(ws_1) \models_\vartheta (h \vdash_{S_1} u_1) \qquad (ws_2) \models_\vartheta (S_\alpha\ h \vdash_{S_2} u_2)}{(subst\ ws_1\ ws_2) \models_\vartheta (h \vdash_{S_2} \mathrm{subst}_\alpha\ 0\ u_1\ u_2)}\ \textsc{WsSubst}$$

$$\frac{(ws) \models_\vartheta (h + [\![\ bs\ ]\!]_\vartheta \vdash_S lift\ u\ [\![\ bs\ ]\!]_\vartheta)}{(strengthen\ ws\ bs) \models_\vartheta (h \vdash_S u)}\ \textsc{WsStrengthen}$$

Figure 8.7: Well-scopedness proof terms

specification in the witness terms always denote the local scope relative to the outer scope h_0.

The rule WnHyp refers to hypotheses in H. The hypothesis for a global variable g represents the assumption that the de Bruijn index $\vartheta\ g$ is well-scoped in the outer scope h_0. The remaining rules axiomatically encode properties of well-scopedness relations. For example, rule WsVar corresponds to the variable rule of semantic well-scoping and WnInvVar to its inversion, i.e. we can establish the well-scopedness of a de Bruijn index from the well-scopedness of a variable constructor. Rule WsSubst denotes the substitution lemma for well-scoping, and WsStrength denotes the strengthening lemma, i.e. an inversion of the weakening lemma.

Since the axiomatic rules of the witness language are backed by concreted lemmas for well-scoping, we can establish soundness of the interpretation

Lemma 16 (Soundness)**.** *Let* $\vartheta = \overline{(g \mapsto n)}, \overline{(t \mapsto u)}$ *and* $\overline{g'} \subseteq \overline{g}$*. If the hypotheses* $H = \overline{(g' : \alpha)}, \overline{([bs]sym : S)}$ *are valid, i.e.* $\overline{h_0 \vdash_\alpha \vartheta\ g'}$ *and* $h_0 + [\![\ bs\ ]\!]_\vartheta \vdash_S [\![\ bs \mid sym\ ]\!]_\vartheta$*, then*

1.

$$\frac{(wn) \models_\vartheta (h \vdash_\beta n')}{h \vdash_\beta n'}$$

2.

$$\frac{(ws) \models_\vartheta (h \vdash_{T'} u')}{h \vdash_{T'} u'}$$

8.2.2 Proof Elaboration

We can split the proof of the induction steps of the lemma into two stages. First, we establish well-scopedness of terms represented by sort and global meta-variables of the conclusion, potentially by using inversion lemmas on the induction hypotheses. Second, we use the rules of the well-scopedness relations and derived rules to establish well-scopedness of the terms in the conclusion. In short, this step amounts to using the fact that evaluation of well-scoped symbolic expression in a well-scoped context yields well-scoped terms.

Similarly, we can also split the elaboration into two corresponding stages. The first stage elaborates one-hole contexts [Huet, 1997] that describe invertible paths to meta-variable in the premises into witnesses that use inversions and hypotheses. We collect the result of the first stage into an environment $P ::= \overline{g : wn}, \overline{s : ws}$ that holds the proof terms for the meta-variables. The second stage elaborates the expression in the conclusion into witness terms by using P for occurring meta-variables.

We only present the formal elaboration of the second stage. Figure 8.8 contains the elaboration function symws from symbolic expressions to witnesses ws. The argument P is the proof term environment for sort and global meta-variables, and the bs argument is the local scope of the symbolic expressions that is elaborated.

In case of a sort meta-variable or a variable constructor with a global variable we look up the proof term in P. For the global variable we need to weaken to the local scope first and then wrap it in the variable constructor. For a locally bound variable b the helper function symwn_α first uses *local* to witness the fact that b is well-scoped in bs, b and then weakens with the difference bs' to create a proof term for well-scopedness in bs, b, bs'[2].

For a regular constructor K we symbolically evaluate the local scopes of the sort fields and proceed recursively and wrap the results in *reg*. In case of

[2]For simplicity, the presented elaborations in Figure 8.8 have the wrong associativity when weakening local and global meta-variables into the local scope. Loom and Needle contain elaborations that deal with associativity correctly.

$$
\begin{array}{l}
\boxed{\text{symwn}_\alpha : bs \to b \to wn} \\
\quad \text{symwn}_\alpha\ (bs, b, bs')\ b = weaken\ (local\ \alpha\ bs)\ bs' \\
\boxed{\text{symws} : P \to bs \to sym \to ws} \\
\quad \text{symws}\ P\ bs\ s = ws\ \textbf{where}\ (s : ws) \in P \\
\quad \text{symws}\ P\ bs\ (K\ g) = var\ K\ (weaken\ wn\ bs) \\
\qquad \textbf{where}\ (g : wn) \in P \\
\quad \text{symws}\ P\ bs\ (K\ b) = var\ K\ (\text{symwn}_\alpha\ bs\ b) \\
\qquad \textbf{where}\ K : \alpha \to S \\
\quad \text{symws}\ P\ bs\ (K\ \overline{b'}\ \overline{sym}) = reg\ K\ \overline{(\text{symws}\ P\ bs'\ sym)} \\
\qquad \textbf{where}\ K : ([bs_b]b : \alpha) \to ([bs_t]t : T) \to S \\
\qquad\qquad \overline{[(\overline{b \mapsto b'}, \overline{t \mapsto sym})]bs_t \Downarrow bs'} \\
\quad \text{symws}\ P\ (bs, bs')\ (weaken\ sym\ bs') = \\
\qquad weaken\ (\text{symws}\ P\ bs\ sym)\ bs' \\
\quad \text{symws}\ P\ bs\ (subst\ b\ sym_1\ sym_2) = \\
\qquad subst\ (\text{symws}\ P\ bs\ sym_1)\ (\text{symws}\ P\ (bs, b)\ sym_2)
\end{array}
$$

Figure 8.8: Well-scopedness of de Bruijn terms

a symbolic weakening we need to strip off the tail bs' to get the local scope for the recursive position. Finally, for $subst\ b\ sym_1\ sym_2$ we recurse into sym_1 with the same local scope but account for the additional variable b when recursing into sym_2.

8.3 Shifting and Substitution

The last semantic boilerplate lemmas that we consider in this chapter are shifting and substitution lemmas of user-defined relations. For formal proofs we can apply the methodology of this chapter: elaboration into a sound witness language. The witness language in this case also observe term equalities similar to Section 8.1. We do not present the final elaboration but only present all the necessary ingredients to develop it. We first discuss shifting lemmas in Section 8.3.1 and then substitution lemmas in Section 8.3.2.

8.3.1 Shifting

As in the previous sections, we first discuss the necessary proof steps semi-formally and subsequently sketch the formal proof elaboration for shifting lemmas.

Shifting introduces a new binding in the environment of a relation and adapts the relation indices by term-level shifting. In its most generic form the insertion happens between two parts u_1 and u_2 of the environment. In this case, term-level shifting is done using the domain of u_2 as cut-off. The proof proceeds by induction on the derivation of a relation. For the inductive step of each rule, be it a variable rule or a regular rule, we want to apply the same rule again. This may require massaging the proof goal with commutation lemmas of shifting and substitution, i.e. we have to push the global shifting into local weakenings and local substitutions to recreate the symbolic structure of the rule.

More specifically, for a rule with conclusion $R\ \overline{sym}$ and values ϑ the goal of the induction step is to show

$$R\ \overline{\mathrm{shift}_\alpha\ c\ [\![\ \epsilon \mid sym\ ]\!]_\vartheta}.$$

To use rule r again, we need to match the same symbolic structure $\overline{sym}$, i.e. we have to find ϑ' such that the following holds

$$\overline{\mathrm{shift}_\alpha\ c\ [\![\ \epsilon \mid sym\ ]\!]_\vartheta} = \overline{[\![\ \epsilon \mid sym\ ]\!]_{\vartheta'}}. \tag{8.1}$$

This is just $\vartheta' = \overline{(r \mapsto \mathrm{shift}_\alpha\ c\ (\vartheta\ r))}, \overline{(t \mapsto \mathrm{shift}_\alpha\ (c + [\![\ bs_t\ ]\!]_\vartheta)\ (\vartheta\ t))}$ or in other words: shifting commutes with symbolic evaluation.

Similar to Section 8.2, we can give a formal syntax-directed elaboration for equation (8.1) from symbolic expressions into a domain-specific witness language of term equalities. This is an extension of the language of Section 8.1 that additionally has primitives for the commutation lemmas. After using rule r we are still left with it's premises as proof goals. In case of a judgement, we need to apply equality (8.1) in the opposite direction.

8.3.2 Substitution

Much of the reasoning is similar to shifting: for context-parametric regular rules we need to commute substitution with symbolic evaluation, which is done by elaboration to term equality witnesses. In the case of a homogenous substitution, the lookup of a variable rule is replaced by a derivation tree of the relation and in the case of a heterogeneous substitution, the lookup is

adapted and wrapped inside the variable rule again. Hence we only discuss the problematic case of non-context parametric regular rules. As an example consider the algorithmic transitivity rule of of *System F with sub-typing*:

$$\frac{(\alpha <: \sigma) \in \Gamma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash \alpha <: \tau} \quad .$$

In the inductive step of the type-variable substitution lemma, e.g. when substitution a type-variable β for σ', we cannot apply the same rule again, because in the case $\alpha = \beta$ it has been substituted. However, we can substitute the lookup by a derivation and use the induction hypothesis of the second premise, i.e. it suffices to show

$$\frac{\Gamma' \vdash [\beta \mapsto \sigma']\alpha <: [\beta \mapsto \sigma']\sigma) \qquad \Gamma' \vdash [\beta \mapsto \sigma']\sigma <: [\beta \mapsto \sigma']\tau}{\Gamma' \vdash [\beta \mapsto \sigma']\alpha <: [\beta \mapsto \sigma']\tau}$$

where Γ' is the resulting context after substitution. In other words, we have spliced in derivations for lookups not only in the case of a variable constructor, but also for lookups in non-context parametric rules. We can finish the proof of the induction step, if we can show that the declarative sub-typing rule is admissible:

$$\frac{\Gamma \vdash \rho <: \sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash \rho <: \tau} \quad .$$

Unfortunately, the admissibility of this rule is a meta-theoretic property of the sub-typing relation that has to be proved separately and that we cannot establish automatically. As a consequence, for this sub-typing relation and similar languages with non-context parametric rules we cannot establish the substitution lemma generically.

However, the above gives us a recipe to automatically derive sufficient proof obligations to establish the inductive step. Consider the substitution lemma for namespace α of a relation R and let S be the sort for namespace α. For each non-context parametric rule of R, we derive a proof obligation by replacing all occurrences of global meta-variables $\overline{g}$ of namespace α with fresh sort meta-variables $\overline{s_g}$ of sort S and each lookup premise $\{g \texttt{ -> } \overline{sym}\}$ with a judgement premise $([\epsilon]R\ (s_g, \overline{[g \mapsto s_g]sym}))$. Note that $[g \mapsto s_g]$ denotes a substitution at the meta-level and not the object level or a symbolic substitution. This substitution needs to ensure that well-scoping of symbolic expression is not violated: each occurrence of g at local scope bs is in fact replaced by (**weaken** s_g bs).

Global meta-variables of other namespaces and lookups of other clauses are left unchanged except for the meta-substitution $[g \mapsto s_g]$ in the lookup data.

8.4 The Loom Generic Library

We implemented the datatype-generic library Loom around Knot specifications that provides boilerplate functionality. Loom is only built for the development of elaborations without end users in mind. Hence direct usability is not a concern. The universe of the library covers generic representations of sorts, environments and expressions, since these determine all of the interesting elaborations. The universe does not deal with user-defined relations and their derivation trees.

Following our free monad principle, we capture de Bruijn terms in a free monadic structure similar to the one in Section 6.2.1. We use the universe of finitary containers [Abbott et al., 2003; Gambino and Hyland, 2004; Jaskelioff and Rypacek, 2012; Moggi et al., 1999] to model the underlying pattern functors of regular constructors of sorts, in order to deal with any positivity and termination requirements. Finitary containers closely model our specification language: a set of shapes (constructors) with a finite number of fields. Each field is associated with a binding specification, all constraints for the well-formedness of specifications and the well-scopedness of meta-variables are encoded in the universe codes using strong intrinsic types [Benton et al., 2012]. Furthermore, we use an indexed [Altenkirch and Morris, 2009] version of finitary containers to model mutually recursive types and use a higher-order presentation to obtain better induction principles for which we assume functional extensionality [3].

The boilerplate lemmas implemented in the library follow the elaboration methodology outlined in this chapter. In total, the library consists of about 4.3k lines of Coq code.

8.5 The Needle Code Generator

Needle is a Code generator written in about 11k lines of Haskell. Needle takes a Knot specification and generates Coq code for the representation of the syntactic sorts of an object language and its semantic relations. Furthermore, it generates specialized language-specific non-generic boilerplate definitions, lemmas and proofs.

[3] However, the code based on our generator Needle does not assume any axioms.

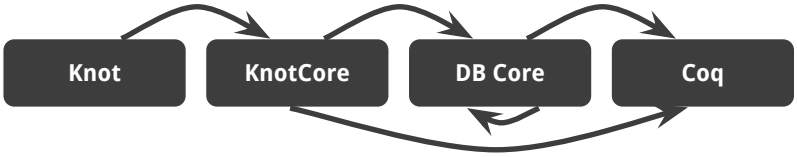


Figure 8.9: Needle Processing Stages

Needle processes Knot specifications in multiple stages which is pictured in Figure 8.9.

The first stage is the parsing and checking of a textual Knot specification, name-resolution, sort checking, scope inference and scope checking, and grouping of mutually recursive sorts and functions. The result is a KnotCore specification that is enriched with the inferred information such as the scope of binding meta-variables. Furthermore, information is also cached in different parts of the specification for easy access. For instance, each symbolic sub-expression is annotated with its scope.

The elaboration to Coq code is performed both, directly from KnotCore and via the intermediate representation DbCore. DbCore consists of a generic representation of symbolic de Bruijn terms and the domain-specific witness languages from this chapter. Needle first generates generic version of boilerplate in DbCore, for example, of lemma statements. The specialization to the given concrete specification is then carried out by a simplifier in DbCore. The simplification steps mainly consist of applying sub-ordination information and partially evaluating the symbolic terms.

The elaborated proof terms tend to be very large. For almost all cases simpler specialized proofs exists. For example, in the case of empty binding specifications some proof steps are entirely unnecessary, however, they are still part of the elaboration. To speed up the checking of the generated code DbCore's simplifier also performs simplification of the domain-specific witness terms. For that, it uses the group structure of equality proofs [Stump and Tan, 2005] with transitivity as the group operation, reflexivity as the neutral element and symmetry as the inversion. Furthermore, the simplifier performs partial evaluation of lemmas that have been added as axioms to the witness languages, e.g. EqhAssocPlus in Figure 8.2[4] The correctness of the proof simplification has also been verified in Loom.

[4]In contrast to the witness languages presented in this chapter, DbCore's representation contains symbolic terms that exactly describe the left and right hand sides of the witnessed equality. This extra information is needed for the partial evaluation.

The Coq code produced by Needle contains both, elaboration proof terms and invocations of proof scripts that are implemented in an accompanying library. To reduce the development effort, lemmas that do not directly depend on the structure of user-defined sorts and relations are primarily implemented via proof scripts, since for this kind of lemmas, proof scripts are sufficiently robust. These include lemmas about domains, indices and contexts, but also the variable cases of interaction lemmas which are always proven separately. Lemmas that induct over sorts or relations directly depend on the user-defined structure and are therefore more brittle. These are always implemented via by proof term elaboration.

8.6 Related Work

There is plenty of existing work on derivation of variable binding boilerplate. We focus only on related work that target mechanization of meta-theory in proof-assistans.

The approaches to tackle the binding boilerplate in mechanizations generally fall in one of several categories: meta-languages with built-in support for binding [Gacek, 2008; Pientka and Dunfield, 2010; Pfenning and Schürmann, 1999; Urban and Tasson, 2005] that also take care of boilerplate automatically, code-generators that produce code for proof-assistants [Aydemir and Weirich, 2010; Polonowski, 2013], libraries that use reflection methods to derive code [Schäfer et al., 2015b], datatype-generic programming libraries [Anand and Rahli, 2014; Lee et al., 2012] that implement boilerplate for a generic universe of datatypes or libraries of reusable tactics [Chlipala, 2008; Pottier, 2013].

Logical frameworks The advantage of logical frameworks based on higher-order abstract syntax is that facts about α-equivalence and well-scoping and structural properties for substitution, weakening and exchange are inherited from the meta-language and thus do not need to be proved separately. Substitution lemmas for semantical relations are also provided for free if the object-language context admits exchange. If it does not admit exchange, there are techniques like explicit contexts [Project, 2015; Lee et al., 2007; Pientka and Dunfield, 2008] that can be used to model relations which then require a manual proof of the substitution lemma while still reaping all the benefits for the syntax. This is, for example, necessary to model dependently typed languages and also for the POPLmark challenge to isolate a variable in the middle of the context for narrowing. In comparison, our derivation of substitution lemmas is independent of any exchange property of contexts.

Decision Procedures The authors of the AutoSubst library [Schäfer et al., 2015b] have developed a language with symbolic expressions for two sorts: de Bruijn terms and simultaneous de Bruijn substitutions with an axiomatic equivalence [Schäfer et al., 2015a] based on the reduction rules of the σ-calculus [Abadi et al., 1991a] which they show to coincide with the semantic equivalence on de Bruijn terms. They have also developed a decision procedure, that automatically decides the equivalence of two expressions. As a consequence, frameworks based on simultaneous de Bruijn substitutions such as AutoSubst can, in theory, derive all the necessary rewrites for the inductive steps of the substitution lemma automatically. In contrast, we derive the rewrites using syntax-directed elaboration. However, the decision procedure of [Schäfer et al., 2015a] is applicable in instances other than substitution lemmas.

Traversal Abstraction Allais et al. [2017b] go beyond the definition of syntactic traversals and look at the definition of a recursor on well-scoped simply-typed λ-terms, which allows the scope-preserving definition of semantic functions like CPS-transformation and evaluation. They prove generic simulation and fusion laws for their recursor, but fall short of deriving interaction lemmas. This work was subsequently extended by Kaiser et al. [2018] for multi-variate languages.

Instead of abstracting over the structure of datatypes like Loom, it is also possible to abstract at another level, for instance over the types themselves and generic traversal functions. This is the approach taken by Pottier [2013] and Keuchel [2016]. Boilerplate lemmas such as interaction lemmas are derived generically from key properties of the traversal. The provided information is however not sufficient to derive other types generically, such as for example well-scoping predicates or symbolic expressions. The traversal properties required by Keuchel [2016] are essentially the fusion and simulation laws of [Allais et al., 2017b], albeit specialized to "syntactic semantics".

DGP for abstract syntax The application of datatype-generic programming techniques to represent abstract syntax with binding and provide generic functionality is quite appealing and has been investigated before. Licata and Harper [2009] and Keuchel and Jeuring [2012] define universes for datatypes with binding in Agda using a well-scoped de Bruijn representations. Loom and Keuchel and Jeuring [2012]'s index universe codes by namespaces and syntactic sorts and interpret it as one big mutually recursive family of types. However, the specification of bindings is different. Keuchel and Jeuring [2012] use a set of abstraction primitives while Loom explicitly models expressions

for binding specifications.

Lee et al. [2012] develop the GMeta framework for first-order representations of variable binding in Coq that supports both locally nameless and de Bruijn representations and generically implement boilerplate lemmas for use in mechanizations. GMeta's universe is based on regular tree types [Morris et al., 2006]. For multi-sorted languages, the implementation of substitution relies on checking for equal sort representations of a variable's sort and a substitute's sort. This check is necessary since subordinate sorts are modeled by providing their full representation in recursive positions. In Loom, the indexing of universe codes by namespaces and sorts provides this information and a simple index check is sufficient. As a result, Loom does not rely on decidable equality of representations and avoids the adequacy issue in the admittedly unlikely case that multiple sorts have equal representations.

Essentially, Loom includes more information of interest explicitly in its codes, i.e. the direct access to recursive positions of subordinate sorts[5] Loom's approach is akin to creating a multi-argument pattern functor for each sort by abstracting over each subordinate sort with variables, or more precisely abstracting over namespaces, and implementing substitutions using the generic functorial map of the corresponding namespace. In GMeta, this information is not explicit, i.e. the functorial abstraction is not possible, and is regained during substitution by basically implementing dynamic typing using structure representations [Abadi et al., 1991b; Weirich, 2006][6]

Allais et al. [2017a] define a universe for languages with a single namespace. Similar to Loom and GMeta, they look at the syntax freely generated by a pattern functor, and adapt the universe of [Chapman et al., 2010] for the functorial abstraction. They generically define a well-scoped recursor [Allais et al., 2017b] and generically prove fusion and simulation laws for it.

Nominal Representations Nominal Isabelle [Urban and Tasson, 2005] is an extension of the Isabelle/HOL framework based on nominal logic [Pitts, 2003] with support for nominal datatype terms which provides reasoning modulo α-equivalence for free. It is also possible to define and reason about semantic relations but for reasoning modulo α-equivalence we first need to show that the relations are equivariant. These equivariance proofs are very similar to the

[5]The access to variables of subordinate sorts is what we are actually interested in. But the result of substituting a variable needs to be joined back in which in Knot always happens at recursive positions.

[6]One can argue that Loom also implements dynamic typing by using type tags. Nevertheless, the indexing of universe codes essentially means that indices live "one level up" compared to structure representations, which grants Loom a more static feel.

proofs of shifting and substitution lemmas for relation in the de Bruijn representation. We hence conjecture that, in a nominal interpretation of KNOT, the binding functions of KNOT are equivariant, i.e. they are binding operators in the sense of [de Amorim, 2016] and that we can generically derive equivariance and substitution lemmas for semantical relations on nominal terms.

8.7 Contributions

The rigorous definition of elaborations and their formal verification in LOOM is a significant leap up from similar existing systems like LNGEN [Aydemir and Weirich, 2010] and DBGEN [Polonowski, 2013], which rely on powerful but brittle tactics to derive boilerplate lemmas.

Furthermore, our approach tackles more boilerplate than any other approach using a first-order representation. To the best of our knowledge, NEEDLE is the first system that derives substitution lemmas for a class of semantic relations.

Chapter 9

Evaluation

This chapter evaluates the benefits of the KNOT approach for type-safety mechanizations with two complimentary evaluations. The first considers different mechanizations for the same language (the POPLMARK challenge) authored by different people with different degrees of automation or tool support. The second compares KNOT against manual mechanizations (written by the same author in a consistent and highly automated style) across different languages.

9.1 Comparison of Approaches

Because it is the most widely implemented benchmark for mechanizing metatheory, we use parts 1A + 2A of the POPLMARK challenge to compare our work with that of others. These parts prove type-safety for System $F_{<:}$ with algorithmic subtyping. As they involve only single-variable bindings, they are manageable for most existing approaches. Figure 9.1 compares 10 different solutions:

1. Charguéraud's [Charguéraud, 2007] developments use the locally-nameless representation and come with automation via proof tactics for this representation.

2. Vouillon [Vouillon, 2012] presents a self-contained de Bruijn solution. The solution only moderately uses automation and instead performs proof steps explicitly for didactic purposes.

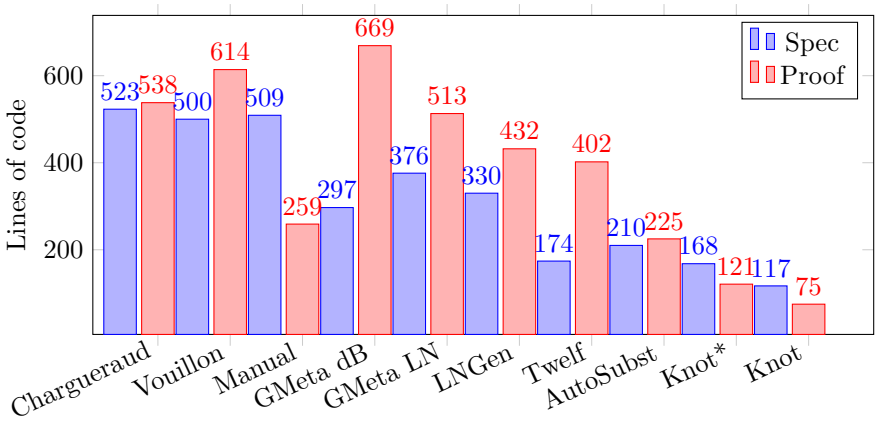


Figure 9.1: Sizes (in LoC) of POPLMARK solutions

3. Our own more compact manual mechanization (see Section 9.2) based on de Bruijn terms with more automation than Vouillon's solution.

4–5. Two solution based on the GMETA [Lee et al., 2012] datatype-generic library for de Bruijn and locally-nameless representations.

6. A mechanization in COQ using a locally-nameless representation and boilerplate produced by the LNGEN [Aydemir and Weirich, 2010] code generator from an OTT specification.

7. A solution in the TWELF logical framework [Ashley-Rollmann et al., 2005].

8. A solution using the AUTOSUBST [Schäfer et al., 2015b] COQ library for de Bruijn terms.

9. A KNOT-based solution without generation of semantics-related boilerplate [Keuchel et al., 2016] (KNOT*).

10. Our KNOT-based solution (KNOT) for both syntax and semantics.

The figure shows the code size separated into *proof* scripts and other *spec*ification lines as generated by *coqwc*, except for the TWELF solution were we made the distinction manually. We excluded both library code and generated code. The AUTOSUBST and KNOT formalizations are significantly smaller

than the others. KNOT's biggest savings compared to AUTOSUBST come from the generic handling of well-scopedness predicates and semantic relations which are not supported by AUTOSUBST. In comparison to the KNOT-based solution [Keuchel et al., 2016] without support for relations, we save relatively more LoC in *proofs* than in *specifications*. In summary, the KNOT solution is the smallest solutions we are aware of.

9.2 Manual vs. Knot Mechanizations

The previous comparison only considers the type-safety proof for a single language, and thus paints a rather one-sided picture of the savings to be had. For this reason, our second comparison considers the savings across 11 languages. As their mechanizations are not readily available across different tools and systems, we here pit KNOT & NEEDLE only against our own manual mechanizations. To yield representative results, all our manual mechanizations have been written by the same author in a consistent and highly automated style.[1]

The 11 textbook calculi we consider are: 1) the simply typed lambda calculus λ, 2) λ extended with products, 3) System F, 4) F with products, 5) F with existentials, 6) F with existentials and products, 7) F with sequential lets, 8) $F_{<:}$ as in the POPLMARK challenge 1A + 2A of Section 9.1, 9) $F_{<:}$ with products, 10) λ with type-operators, and 11) F with type-operators.

For each language, we have two COQ formalizations: one developed without tool support and one that uses NEEDLE's generated code. Table 9.1 gives a detailed overview of the code sizes (LoC) of different parts of the formalizations and the total and relative amount of boilerplate. It also shows the ΔNEEDLE savings due to NEEDLE's new support for relations.

The *Specification* column comprises the language specifications. For the manual approach, it is split into an *essential* part (Ess.) and a *boilerplate* part (Bpl.). The former comprises the abstract syntax declarations (including binding specifications), the evaluation rules, typing contexts and typing rules; this part is also expressed (slightly more concisely) in the KNOT specification (NDL). The boilerplate part consists of lookups for the variable rules and shifting and substitution functions, that are necessary to define β-reduction on terms and, where applicable, on types; this boilerplate is generated by NEEDLE and thus not counted towards the size of the KNOT-based mechanization.

The *Metatheory* column shows the effort to establish type-safety of the languages. The essential lemmas are canonical forms, typing inversion, progress

[1] E.g., compare our manual solution against Vouillon's in Figure 9.1: it is smaller due to more automation and simpler definitions that are more amenable to proof search.

		Specification			Metatheory				Total				
		Ess.	Bpl.	NDL	Ess.	Syn.	Sem.	NDL	Man.	NDL		ΔNEEDLE	
1)	λ	41	39	36	23	22	23	19	148	55	(37.2%)	28	(18.9%)
2)	$\lambda_\times$	82	67	75	32	61	72	75	314	150	(47.8%)	48	(15.3%)
3)	F	51	102	46	185	79	30	30	447	76	(17.0%)	42	(9.4%)
4)	$F_\times$	90	150	85	230	126	79	87	675	172	(25.5%)	97	(14.4%)
5)	$F_\exists$	71	114	66	266	86	44	43	581	109	(18.8%)	79	(13.6%)
6)	$F_{\exists,\times}$	123	164	103	365	172	98	101	922	204	(22.1%)	106	(11.5%)
7)	F_{seq}	99	165	88	249	162	49	70	724	158	(21.8%)	89	(12.3%)
8)	$F_{<:}$	66	117	57	264	150	163	138	760	195	(25.7%)	94	(12.4%)
9)	$F_{<:,\times}$	110	155	92	311	212	256	235	1044	327	(31.3%)	149	(14.3%)
10)	λ_ω	97	88	75	202	141	251	268	779	343	(44.0%)	161	(15.4%)
11)	F_ω	120	98	92	204	141	311	330	874	422	(48.3%)	160	(15.3%)

Table 9.1: Size statistics of the meta-theory mechanizations.

and preservation and where applicable this also includes: pattern-matching definedness, reflexivity and transitivity of subtyping and the Church-Rosser property for type reductions.

There are two classes of binder boilerplate in the metatheory:

1. Syntax-related boilerplate (Syn.) that consists of interaction lemmas, weakening and substitution lemmas for well-scopedness predicates. The code size for these lemmas depends mainly on the number of namespaces, the number of syntactic sorts and the dependency structure between them, which is roughly the same for these languages. NEEDLE derives this boilerplate completely automatically.

2. The semantics-related boilerplate (Sem.) are well-scoping, shifting and substitution lemmas for user-defined semantic relations. The size of these lemmas depends on the number of namespaces and the size of the semantics relations.

   We defined λ_ω and F_ω using algorithmic type equality with reflexivity only on type-variables, not on types. For the substitution lemmas of these calculi, NEEDLE generates a proof obligation for the admissibility of reflexivity of types. Similarly, $F_{<:}$and $F_{<:,\times}$use algorithmic subtyping and both, the type-variable reflexivity and type-variable transitivity rule, give rise to a proof obligation. Furthermore, neither of these two rules meets the restrictions of Section 6.5.1 for variable rules, and thus deriving a variable rule yields another proof obligation.

The final column group contrasts the total sizes of the manual (Man.) and KNOT based formalizations (NDL). The last column ΔNEEDLE shows the difference between KNOT-based solutions with and without generation of semantics-related boilerplate.

Summary Table 9.1 clearly shows that NEEDLE provides substantial savings in each of the language formalizations. On average, NEEDLE-based solutions are 71% smaller than the manual solutions (100% − NDL/Man.), the generation of semantics-related boilerplate saves ∼14% of the manual approach (ΔNEEDLE/Man.) and ∼33% of the NEEDLE-assisted approach (ΔNEEDLE/NDL).

Chapter 10

Conclusion

In this concluding chapter we revisit the research question, summarize the results of this thesis and evaluate the contributions in the context of the research question. Furthermore, we highlight opportunities for future work and speculate about how the contributions of this thesis can benefit the programming language research community at large in the future.

10.1 Research Question

Widespread formalization and mechanization of programming language meta-theory is hindered by the large development costs. For further adoption, reducing the costs is crucial. This leads us back to the research question laid out at the start of this thesis:

> **How can we reduce the cost of mechanising the formal meta-theory of programming languages?**

This thesis promotes reuse as a way to reduce the mechanization effort of programming language meta-theory and investigates the two principled approaches *modularity* and *genericity* to reuse in general, and the application of these approaches to programming language theory in particular.

10.2 Summary

This section gives and overview of the thesis and its most important results. We discuss the two parts of the thesis in turn.

10.2.1 Modularity

Part I pursues the modularity approach and universal method for modularization of functional programs on inductive datatypes and modularization of induction proofs for properties of these programs. Specifically, this thesis has contributed new results for the modular representation of datatypes and for reducing feature interaction in effectful semantics. Both contributions have been evaluated with case studies. We discuss both contributions and the case studies below.

Modular Representation

Modularizing proofs is particularly challenging since the proof-assistant settings imposes several restrictions for logical consistency. This challenge is addressed in work prior to this thesis in the *Meta-Theory à la Carte (MTC)* [Delaware et al., 2013] framework which uses Church encodings to represent inductive datatypes and families. This thesis develops an alternative approach, using datatype-generic representations, that improves upon MTC in terms of adequacy, convenience and compatibility with classical logic.

Feature Interaction

A concern in modularization is the interaction between two or more features. Each new feature that is integrated potentially interacts with all previous ones. As a result, extending existing developments exhibits a quadratic increase in effort. This is not a problem that is specific to a modular approach, but applies to software development in general. However, it is an obstacle to modularity if the interaction involves the complete set of features instead of an isolated subset, e.g. two features.

Problematic feature interactions appear in the semantics of programming languages with effectful features. To cut down this particular kind of interaction, we developed a denotational semantics based on monad transformers and corresponding algebraic laws. This semantics has allowed us to to formulate and prove a general type-soundness theorem in a modular way that enables the modular reuse of language feature proofs and reduce the interaction between effectful language features to the interaction between their effects.

Case Studies

We have performed two case studies to evaluate our contributions. The first case study is a port of the MTC case study to the container based modular representation. It consists of continuity and type-safety proofs for seven language features and six language compositions. In contrast to MTC, the preservation of the universal properties is not necessary for induction which makes the approach conceptually simpler. This is reflected in the case study: sigma types and its projections are removed from lemmas and proofs, and language features and their (proof) algebras are on average 240 LoC (25%) smaller. The final language compositions are slightly larger by about 6 LoC (7%), but are short in comparison to the feature specific code.

The second case study demonstrates the 3MT framework. It consists of five reusable language features: pure boolean and arithmetic features and effectful features for references, errors and lambda abstractions. The study builds 28 different combinations of the features. Each language feature has its own reusable feature theorem and each of the six effect combinations its own effect theorem. Effect theorems are only reusable for languages with the specific set of effects. Like in the first case study, the feature and effect theorems outweigh the final language compositions.

10.2.2 Genericity

Part II examines the genericity approach for a specific use case: variable binding boilerplate in mechanizations. A key ingredient of reduction-based semantics of programming languages is substitiution. Meta-theoretic proofs using these semantics usually involve a large number of burdensome boilerplate proofs about substitutions which distracts a human semanticist from essential theorems when mechanising her proofs. This thesis develops a generic solution to the boilerplate lemmas based on a novel specification language Knot for programming languages and a code generator Needle that produces boilerplate code. We summarize and discuss the contributions.

Specification Language

Knot allows the specification of abstract syntax, with explicit specifications of binding, and of semantical relations between syntax terms. Relations are defined using arbitrary expressions built from a language's abstract syntax. Knot employs a novel type system that checks that all expressions, including substitutions that may appear in them, are always well-scoped.

Free Monadic Structure A central contribution of this thesis is the identification of a large class of syntaxes for which boilerplate is completely generically derivable: syntactic sorts that have a free monad-like structure. For relations this translates to context parametricity of regular (non-variable) rules.

Principled Elaboration NEEDLE produces specialized definitions for a given KNOT specification in the COQ proof assistant and produces code for boilerplate functions and boilerplate lemmas. We employ a principled approach to elaboration of boilerplate code that gives us confidence in the correctness of our code generator: we have developed and formally verified elaborations in the COQ proof assistant using our datatype-generic implementation LOOM of KNOT.

Case Study Our case study compares our generic approach against fully manual mechanizations of type safety proofs of 10 lambda calculi. It shows substantial savings in the mechanization for all 10 calculi with the largest savings being about 74% reduction in code size for System F. This case study indicates that replacing manual variable binding boilerplate by reusable generic solutions is indeed an effective means of reducing the mechanization effort.

10.3 Future Work

This section presents possible directions for further research. We group the ideas around the two parts of the thesis and also discuss the combination of the two parts.

10.3.1 Modularity

We have achieved modular reuse of structurally recursive functions and structural induction proofs. This is a crucial first step towards reducing mechanization effort using the modularity methodology. However, the focus was on feasibility. Further research is necessary to make the approach more convenient and practical. Furthermore we targeted a specific class of languages (simply-typed and effectful lambda calculi), specific semantics ((monadic) denotational semantics) and specific meta-theoretic properties (type safety). Of course the scope can be extended in either of these directions. We discuss the most promising ideas for increasing the convenience and scope below.

Convenience

Our case study contains a substantial amount of bookkeeping of the relationship of final and intermediate compositions of datatypes, relations, algebras and proof algebras and similarly between the final and intermediate compositions of effects for our monadic denotational semantics. Most of this is of course boilerplate code that should be taken care of automatically.

Immediate and easy is to provide better specialized proof automation, for instance for type class derivations and algebra dispatch proof steps. Furthermore, a lot of the repetitive abstractions over super functors and algebras can be dealt using with available modularity features in proof assistants like modular, module functors and canonical substructures.

Ultimately, the support for modular induction proofs can be built into proof assistants to make this method practical on a larger scale. Concepts like *proof algebras* presented in this thesis can be turned into first-class primitives of the proof-assistant's language. The generic universe implementation of the first part can be used as a basis for an elaboration into a core calculus.

Scope

As previously discussed, our case studies covered type safety proofs for simply-typed lambda calculi with a (monadic) denotational semantics. We discuss different possible extensions.

An obvious extension would be to investigate the modularizability and feature interaction of richer language, be it with more expressive type systems, e.g. involving qualified types, polymorphism, dependent types or sub-typing, with more complex language features like datatypes or modules, or with a different principal programming paradigm like object-oriented or logical programming language calculi.

Another extension of our work, is to modularize other meta-theoretic proofs than type safety. As laid out in the introduction of the thesis, an important direction is verified compilation of programs. This can be combined with the modularity methodology: write compilers that are modular in the source – or even source and target language – and modularly verify semantically correct compilation.

Other interesting proofs are logical relation-based proofs of type safety, strong normalization, parametricity or full-abstraction. Logical relations are defined by structural recursion on the type structure of languages and should therefore be easy to modularize. However, feature interactions in the proofs

still depend on the language features and it remains to be seen how easy it is to modularize the interactions.

Semantics In our case studies we modularized type safety proofs and reduced non-modular feature interaction between effectful features using a monadic denotational semantics. Monads are, however, not the only approach to model side-effects and it would be interesting to see how modularizable alternative approaches are. Since our monadic typing rules with explicit continuations are reminiscent of algebraic effects and handlers [Bauer and Pretnar, 2015; Plotkin and Pretnar, 2009; Kammar et al., 2012], this is one particularly promising alternative to model effects. Furthermore, since algebraic effect handlers have better composability properties than monads, they potentially allow further reduction of feature interaction.

10.3.2 Genericity

Scope

There are multiple possibilities to grow the Knot specification language to handle more object languages. For instance, we can extend Knot with support for programming and let the user write functions. These functions could then also be used in the definition of relations by extending the expression language accordingly. Additionally, we can import more concepts like GADTs and derive boilerplate for intrinsically well-typed syntax and include dependent types in the expression language.

Furthermore, we can integrate more variable binding like concepts into Knot, for instance, first-class substitutions. This will also require us to improve Knot scope-checking type system.

The Needle code generator can be scaled to include the above extensions. However, Needle can also be scaled independently. Currently, the code is geared towards type safety proofs, but other meta-theoretic proofs may require different boilerplate that could be generated by Needle. Furthermore, Knot itself is independent of a particular representation. Hence, we can imagine generating boilerplate for different representations like a nominal, locally-named or locally-nameless representation.

Mathematical Foundations

Last but not least, it would be interesting to explore the mathematical foundations that underly Knot. Categorical models exist for syntax with variable

binding, but KNOT's features exceed what can be described in these models. For instance, the usual limitation is that only one namespace is assumed.

A promising area is the modelling of well-scoped terms as a generalization of relative monads. This is all the more interesting, since there is already research available that models semantic relations as modules over relative monads.

Appendices

Appendix A

Needle & Knot

A.1 Free Monadic Well-Scoped Terms

The datatype $Free_{Stx}$ in Figure A.1 shows the generic construction of free monadic well-scoped terms from a base functor. Notice that the representation of variables is not fixed to be *Fin* but turned into a parameter for uniformity with $Free_{Set}$.

One of the problems is how to represent morphisms between two families $v\ w :: Nat \to *$ when functorially mapping $f\ v\ d \to f\ w\ d$. In general, we cannot lift a function of type $v\ d_1 \to w\ d_2$ to a function of type $v\ (S\ d_1) \to w\ (S\ d_2)$. We side-step this issue by abstracting away over the representation m of such morphisms and require an interpretation function $\forall d_1\ d_2.m\ d_1\ d_2 \to v\ d_1 \to w\ d_2$ [Keuchel, 2016].

Simultaneous substitutions, i.e. mapping variables to terms, is generically defined in Figure A.2.

```
data Free_Stx (f :: (Nat → ∗) → (Nat → ∗)) v d where
  Return_Stx :: v d → Free_Stx f v d
  Step_Stx   :: f (Free_Stx f v) d → Free_Stx f v d
class Functor_Stx (f :: (Nat → ∗) → (Nat → ∗)) where
  fmap_Stx   :: ∀v :: Nat → ∗ (w :: Nat → ∗) (m :: Nat → Nat → ∗).
                (∀d1 d2.m d1 d2 → m (S d1) (S d2)) →
                (∀d1 d2.m d1 d2 → v d1 → w d2) →
                ∀d1 d2.
                m d1 d2 → f v d1 → f w d2
class Monad_Stx (f :: (Nat → ∗) → (Nat → ∗)) where
  return_Stx ::  v d → f v d
  bind       ::  ∀v :: Nat → ∗ (w :: Nat → ∗) (m :: Nat → Nat → ∗).
                (∀d1 d2.m d1 d2 → m (S d1) (S d2)) →
                (∀d1 d2.m d1 d2 → v d1 → f w d2) →
                ∀d1 d2.
                f v d1 → m d1 d2 → f w d2
instance Functor_Stx f ⇒ Monad_Stx (Free_Stx f) where
  return_Stx = Return_Stx
  bind up int t f = case t of
    Return_Stx x → int f x
    Step_Stx x   → Step_Stx (fmap_Stx up (flip (bind up int)) f x)
```

Figure A.1: Free Monads for Well-Scoped Terms

```
newtype Sub (f :: (Nat → ∗) → (Nat → ∗)) d1 d2 where
  Sub :: {fromSub :: Fin d1 → f Fin d2} → Sub f d1 d2
upSub :: Monad_Stx f ⇒ Sub f d1 d2 → Sub f (S d1) (S d2)
upSub (Sub m) = Sub $ λx → case x of
  FZ → return_Stx FZ
  FS x → subst (m x) (return_Stx ∘ FS)
subst :: Monad_Stx f ⇒ f Fin d1 → (Fin d1 → f Fin d2) → f Fin d2
subst t = bind upSub fromSub t ∘ Sub
```

Figure A.2: Generic Simultaneous Substitution

```
data LamF (v :: Nat → ∗) (d :: Nat) where
  AppF :: v d → v d → LamF v d
  AbsF :: v (S d) → LamF v d
instance Functor_Stx LamF where
  fmap_Stx up int f t = case t of
    AppF t t → AppF (int f t) (int f t)
    AbsF t   → AbsF (int (up f) t)
type Lam′ = Free_Stx LamF Fin
substLam′ :: Lam′ d1 → (Fin d1 → Lam′ d2) → Lam′ d2
substLam′ = subst
```

Figure A.3: Free Monad Instantiation

A.2 Well-scoped Evaluation

Figure A.4 gives the full interpretation of well-scoping proof terms.

Lemma 17. *The syntax-directed elaboration is correct, i.e.*

$$\frac{\mathcal{L}; bs \vdash sym : S \qquad (wn) \models_\vartheta (h_0 \vdash_\alpha n) \qquad (ws) \models_\vartheta (h_0 + [\![\, bs_t \,]\!]_\vartheta \vdash_T u) \qquad P = \overline{\overline{(r : wn), (t : ws)}}}{(symws\ P\ bs\ sym) \models_\vartheta (h_0 + [\![\, bs \,]\!]_\vartheta \vdash_S [\![\, bs \mid sym \,]\!]_\vartheta)}$$

Corollary 18 (Well-scoped evaluation)**.**

$$\frac{\mathcal{L}; bs \vdash sym : S \qquad h_0 \vdash n : \alpha \qquad h_0 + [\![\, bs_t \,]\!]_\vartheta \vdash u : T}{h_0 + [\![\, bs \,]\!]_\vartheta \vdash [\![\, bs \mid sym \,]\!]_\vartheta : S}$$

$\boxed{(wn) \models_\vartheta (h \vdash_\alpha n)}$

$$\frac{(g : \alpha) \in H}{(hyp\ g\ \alpha) \models_\vartheta (h_0 \vdash_\alpha \vartheta\ g)}\ \textsc{WnHyp}$$

$$\frac{}{(zero\ \alpha\ bs) \models_\vartheta (S_\alpha(h_0 + [\![\ bs\ ]\!]_\vartheta) \vdash_\alpha 0)}\ \textsc{WnZero}$$

$$\frac{(wn) \models_\vartheta (h \vdash_\alpha n)}{(weaken\ wn\ bs) \models_\vartheta (h + [\![\ bs\ ]\!]_\vartheta \vdash_\alpha \mathrm{wk}_\alpha\ n\ [\![\ bs\ ]\!]_\vartheta)}\ \textsc{WnWeaken}$$

$$\frac{(wn) \models_\vartheta (h + [\![\ bs\ ]\!]_\vartheta \vdash_\alpha \mathrm{wk}_\alpha\ n\ [\![\ bs\ ]\!]_\vartheta)}{(strengthen\ wn\ bs) \models_\vartheta (h \vdash_\alpha n)}\ \textsc{WnStrengthen}$$

$$\frac{K : \alpha \to S \qquad (ws) \models_\vartheta (h \vdash_S K\ n)}{(varinv\ K\ ws) \models_\vartheta (h \vdash_\alpha n)}\ \textsc{WnInvVar}$$

$\boxed{(ws) \models_\vartheta (h \vdash_S u)}$

$$\frac{([bs]sym : S) \in H}{(hyp\ bs\ sym) \models_\vartheta (h_0 + [\![\ bs\ ]\!]_\vartheta \vdash_S evalsym\ bs\ sym\ \vartheta)}\ \textsc{WsHyp}$$

$$\frac{K : \alpha \to S \qquad (wn) \models_\vartheta (h \vdash_\alpha n)}{(var\ K\ wn) \models_\vartheta (h \vdash_S K\ n)}\ \textsc{WsVar}$$

$$\frac{K : \overline{b : \alpha} \to \overline{[bs]t : T} \to S \qquad \vartheta' = \overline{t \mapsto u} \qquad (ws_i) \models_\vartheta (h + [\![\ bs_i\ ]\!]_{\vartheta'} \vdash_{T_i} u_i) \quad (\forall i)}{(reg\ K\ \overline{ws}) \models_\vartheta (h \vdash_S K\ \overline{u})}\ \textsc{WsReg}$$

$$\frac{K : \overline{b : \alpha} \to \overline{[bs]t : T} \to S \qquad \vartheta' = \overline{t \mapsto u} \qquad (ws) \models_\vartheta (h \vdash_S K\ \overline{u})}{(reginv\ K\ n\ ws) \models_\vartheta (h + [\![\ bs_i\ ]\!]_{\vartheta'} \vdash_{T_i} u_i)}\ \textsc{WsRegInv}$$

$$\frac{(ws) \models_\vartheta (h \vdash_S u)}{(weaken\ ws\ bs) \models_\vartheta (h + [\![\ bs\ ]\!]_\vartheta \vdash_S \mathrm{wk}\ u\ [\![\ bs\ ]\!]_\vartheta)}\ \textsc{WsWeaken}$$

$$\frac{(ws) \models_\vartheta (h + [\![\ bs\ ]\!]_\vartheta \vdash_S \mathrm{wk}\ u\ [\![\ bs\ ]\!]_\vartheta)}{(strengthen\ ws\ bs) \models_\vartheta (h \vdash_S u)}\ \textsc{WsStrengthen}$$

$$\frac{(ws_1) \models_\vartheta (h \vdash_{S1} u1) \qquad (ws_2) \models_\vartheta (S_\alpha\ h \vdash_{S2} u2)}{(subst\ ws_1\ ws_2) \models_\vartheta (h \vdash_{S2} \mathrm{su}_\alpha\ 0\ u1\ u2)}\ \textsc{WsSubst}$$

Figure A.4: Well-scoping proof term interpretation

$$
\begin{array}{l}
qt ::= \text{qrf } bs\ sym\ S \mid \text{qsm } qt \mid \text{qtr } qt\ qt \\
\quad \mid \text{qcn } K\ \overline{qt} \mid \text{qhw } bs \mid \text{qhu} \mid \text{quw} \mid \text{quu}
\end{array}
$$

Figure A.5: Grammar of term equality witnesses

$$
\begin{array}{l}
\boxed{shsym : bs \to sym \to qt} \\
\quad shsym\ bs\ s \qquad = qrf\ bs\ s \\
\quad shsym\ bs\ (K\ r) = qrf\ bs\ (K\ r) \\
\quad shsym\ bs\ (K\ b) = qrf\ bs\ (K\ b) \\
\quad shsym\ bs\ (K\ \overline{b}\ \overline{sym}) = \\
\qquad qcn\ K\ \overline{shsym\ (bs, bs')\ sym} \\
\qquad \textbf{where}\ K : \overline{b' : alpha} \to ([\,bs'\,]\ t : T) \to S \\
\quad shsym\ bs\ (weaken\ s\ bs') = qhh\ bs\ s\ bs' \\
\quad shsym\ bs\ (subst\ b\ sym\ s) = qhu\ bs\ sym\ s
\end{array}
$$

Figure A.6: Shift commutation elaboration

A.3 Relation Shift Elaboration

The shifting lemma for relations require a proof that the global shifting commutes with the local evaluation of expressions. Figure A.5 contains the language of equality witnesses that we are using, Figure A.7 their interpretation, and Figure A.6 shows the elaboration function for the shift commutation lemma.

The relation $qt \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = v : S)$ interprets equality witnesses for the purpose of proving commutation of symbolic evaluation with shifting only. Hence only proof term formers relevant to this lemma are interpreted. The cutoff c and the value environment ϑ are parameters to this relation.

Lemma 19 (Soundness)**.**

$$\frac{eqt \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = v : S)}{u = v}$$

Lemma 20 (Elaboration correctness)**.** *The elaboration for the shift commu-*

tation is correct.

$$\frac{\begin{array}{c}\mathcal{L}; bs \vdash sym : S \qquad eqt = shsym\ bs\ sym \\ h = [\![\ bs\ ]\!]_\vartheta \qquad u = sh\ (c+h)\ [\![\ bs \mid sym\ ]\!]_\vartheta \qquad v = [\![\ bs \mid sym\ ]\!]_{(sh_\mathcal{L}\ (c+h)\ \vartheta)}\end{array}}{eqt \models_{\mathcal{L},h_0,c,\vartheta} (h_0 + h \vdash u = v : S)}$$

Corollary 21.

$$\frac{\mathcal{L}; bs \vdash sym : S \qquad h = [\![\ bs\ ]\!]_\vartheta}{sh\ (c+h)\ [\![\ bs \mid sym\ ]\!]_\vartheta = [\![\ bs \mid sym\ ]\!]_{(sh_\mathcal{L}\ c\ \vartheta)}}$$

Lookup premise We still need to prove that the premises of rule r hold. A lookup premise $\{x \texttt{ -> } \overline{sym}\}$ for an environment clause $\alpha \texttt{ -> } \overline{S}$ gives rise to the proof obligation

$$\frac{(n : [\![\ bs \mid sym'\ ]\!]_\vartheta) \in \Gamma}{(sh\ c\ n : [\![\ bs \mid sym'\ ]\!]_{sh_\mathcal{L}\ c\ \vartheta}) \in \Gamma}$$

which we get from Corollary 21 and by proving

$$\frac{(n : \overline{u}) \in \Gamma, \Delta \qquad c = \mathrm{dom}\ \Delta}{(sh\ c\ n : \overline{sh\ c\ u}) \in \Gamma, \overline{v}, \Delta}$$

by induction over Δ.

Judgement premise A judgement premise *rbs jmt* $\overline{sym}$ gives rise to the proof obligation

$$\Gamma, \overline{v}, \Delta, [\![\ \epsilon \mid rbs\ ]\!]_{sh_\mathcal{L}\ c\ \vartheta} \vdash_R \overline{[\![\ \lfloor rbs \rfloor \mid sym\ ]\!]_{sh_\mathcal{L}\ c\ \vartheta}}$$

which we get from the induction hypothesis

$$\Gamma, \overline{v}, \Delta, sh\ c\ [\![\ \epsilon \mid rbs\ ]\!]_\vartheta \vdash_R \overline{sh\ (c + [\![\ \lfloor rbs \rfloor\ ]\!]_\vartheta)\ [\![\ \lfloor rbs \rfloor \mid sym\ ]\!]_\vartheta}$$

and Corollary 21.

$\boxed{qt \models_{\mathcal{L},h,c,\vartheta} (h \vdash u = u : S)}$

$$\frac{h = [\![\ bs\ ]\!]_\vartheta \qquad u = sh\ c\ [\![\ bs \mid sym\ ]\!]_\vartheta = [\![\ bs \mid sym\ ]\!]_{(sh_{\mathcal{L}}\ c\ \vartheta)}}{(\text{qrf}\ bs\ sym\ S) \models_{\mathcal{L},h_0,c,\vartheta} (h_0 + h \vdash u = u : S)}\ \textsc{Refl}$$

$$\frac{qt \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = v : S)}{(\text{qsm}\ qt) \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash v = u : S)}\ \textsc{Sym}$$

$$\frac{qt_1 \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = v : S) \qquad qt_2 \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash v = w : S)}{(\text{qtr}\ qt_1\ qt_2) \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = w : S)}\ \textsc{Trans}$$

$$\frac{\begin{array}{c} K : \overline{([bs_b]b : \alpha)} \to \overline{([bs_t]t : T)} \to S \\ \vartheta' = \overline{t \mapsto u} \\ \overline{qt \models_{\mathcal{L},h_0,c,\vartheta} (h + [\![\ bs_t\ ]\!]_{\vartheta'} \vdash u = v : T)} \end{array}}{(\text{qcn}\ K\ \overline{qt}) \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash K\ \overline{u} = K\ \overline{v} : S)}\ \textsc{Reg}$$

$$\frac{h' = [\![\ bs\ ]\!]_\vartheta \qquad qt \models_{\mathcal{L},h_0,c,\vartheta} (h \vdash u = v : S)}{\text{qcw}\ qt\ bs \models_{\mathcal{L},h_0,c,\vartheta} (h + h' \vdash wk\ u\ h' = wk\ v\ h' : S)}\ \textsc{CongWeaken}$$

$$\frac{\begin{array}{c} h_1 = [\![\ bs_1\ ]\!]_\vartheta \qquad h_2 = [\![\ bs_2\ ]\!]_\vartheta \\ c_1 = c + h_1 \\ u_1 = \text{sh}_\alpha\ (\text{sh}^*\ c_1\ h_2)\ (\text{sh}^*\ u\ h_2) \qquad u_2 = \text{sh}^*\ (\text{sh}_\alpha\ c_1\ u)\ h_2 \end{array}}{\text{qhw}\ bs_1\ bs_2 \models_{\mathcal{L},h_0,c,\vartheta} (h_0 + h_1 + h_2 \vdash u_1 = u_2 : S)}\ \textsc{ShiftWeaken}$$

$$\frac{\begin{array}{c} h = [\![\ bs\ ]\!]_\vartheta \\ u_1 = \text{sh}_\alpha\ c\ (\text{su}_\beta\ 0\ u\ v) \\ u_2 = \text{su}_\beta\ 0\ (\text{sh}_\alpha\ c\ u)\ (\text{sh}_\alpha\ (c + 1_\beta)\ v) \end{array}}{qhu\ bs \models_{\mathcal{L},h_0,c,\vartheta} (h_0 + h \vdash u_1 = u_2 : S)}\ \textsc{ShiftSubst}$$

Figure A.7: Term equality semantics

Bibliography

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J. J. (1991a). Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416. ↑155, ↑191

Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1991b). Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268. ↑192

Abbott, M., Altenkirch, T., and Ghani, N. (2003). Categories of Containers. In Gordon, A. D., editor, *Foundations of Software Science and Computation Structures*, FoSSaCS'03, pages 23–38. Springer. ↑64, ↑76, ↑188

Abbott, M., Altenkirch, T., and Ghani, N. (2005). Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27. ↑34, ↑56

Abel, A. (2010). MiniAgda: Integrating Sized and Dependent Types. In Bove, A., Komendantskaya, E., and Niqui, M., editors, *Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*. ↑35

Allais, G., Atkey, R., Chapman, J., McBride, C., and McKinna, J. (2017a). A type and scope safe universe of syntaxes with binding, their semantics and proofs. ↑192

Allais, G., Chapman, J., McBride, C., and McKinna, J. (2017b). Type-and-scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 195–207. ACM. ↑191, ↑192

Altenkirch, T., Chapman, J., and Uustalu, T. (2010). Monads Need Not Be Endofunctors. In Ong, L., editor, *Foundations of Software Science and Computational Structures*, FoSSaCS'10, pages 297–311. Springer. ↑138, ↑141

Altenkirch, T., Chapman, J., and Uustalu, T. (2014). Relative Monads Formalised. *Journal of Formalized Reasoning*, 7(1):1–43. ↑138, ↑141

Altenkirch, T. and McBride, C. (2003). Generic Programming within Dependently Typed Programming. In Gibbons, J. and Jeuring, J., editors, *Generic Programming: IFIP TC2 / WG2.1 Working Conference Programming*, pages 1–20. Springer. ↑55, ↑73

Altenkirch, T., McBride, C., and Morris, P. (2007). Generic Programming with Dependent Types. In Backhouse, R., Gibbons, J., Hinze, R., and Jeuring, J., editors, *Datatype-Generic Programming*, SSDGP'06, pages 209–257. Springer. ↑55

Altenkirch, T. and Morris, P. (2009). Indexed Containers. In *Logic In Computer Science*, LICS'09, pages 277–285. IEEE Computer Society Press. ↑61, ↑73, ↑188

Altenkirch, T. and Reus, B. (1999). Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Flum, J. and Rodriguez-Artalejo, M., editors, *Computer Science Logic*, volume 1683 of *LNCS*, pages 453–468. Springer. ↑138, ↑141

Amin, N. and Tate, R. (2016). Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848. ACM. ↑3, ↑14

Anand, A. and Rahli, V. (2014). A Generic Approach to Proofs about Substitution. In *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, LFMTP '14. ACM. ↑190

Ashley-Rollmann, M., Crary, K., and Harper, R. (2005). CMU's solution to the POPLmark challenge. `https://www.seas.upenn.edu/~plclub/poplmark/cmu.html`. Accessed: 2018-01-10. ↑196

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R., and Weirich, S. (2008). Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15. ACM. ↑121, ↑142

Aydemir, B. and Weirich, S. (2010). LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania, Department of Computer and Information Science. ↑157, ↑190, ↑193, ↑196

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. (2005). Mechanized Metatheory for the Masses: The PoplMark Challenge. In Hurd, J. and Melham, T., editors, *Theorem Proving in Higher Order Logics*, pages 50–65. Springer. ↑23

Bahr, P. and Hvitved, T. (2011). Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP'11, pages 83–94. ACM. ↑105

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland. ↑117

Batory, D., Höfner, P., and Kim, J. (2011). Feature Interactions, Products, and Composition. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE'11, pages 13–22. ACM. ↑91

Bauer, A. and Pretnar, M. (2015). Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123. ↑104, ↑206

Benke, M., Dybjer, P., and Jansson, P. (2003). Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic J. of Computing*, 10(4):265–289. ↑46, ↑55, ↑58, ↑73, ↑76

Benton, N., Hur, C.-K., Kennedy, A. J., and McBride, C. (2012). Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159. ↑139, ↑188

Bird, R. S. and Paterson, R. (1999). De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91. ↑138

Böhm, C. and Berarducci, A. (1985). Automatic synthesis of typed -programs on term algebras. *Theoretical Computer Science*, 39(0):135–154. ↑35, ↑36, ↑43

Cameron, N., Drossopoulou, S., and Ernst, E. (2008). A Model for Java with Wildcards. In Vitek, J., editor, *ECOOP 2008 – Object-Oriented Programming*, pages 2–26. Springer. ↑3

Cenciarelli, P. and Moggi, E. (1993). A Syntactic Approach to Modularity in Denotational Semantics. In *Proceedings of the Conference on Category Theory and Computer Science*, CCTCS'93. ↑103

Chakravarty, M. M. T., Ditu, G. C., and Leshchinskiy, R. (2009). Instant Generics: Fast and Easy. Unpublished Draft. ↑73

Chapman, J., Dagand, P.-É., McBride, C., and Morris, P. (2010). The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 3–14. ACM. ↑75, ↑192

Charguéraud, A. (2007). `http://www.charglueraud.org/softs/ln/`. Accessed: 2015-07-02. ↑195

Cheney, J. and Hinze, R. (2002). A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell'02, pages 90–104. ACM. ↑73

Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP'08, pages 143–156. ACM. ↑96, ↑190

Chlipala, A. (2013). *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. ↑34

Clément, D., Despeyroux, T., Kahn, G., and Despeyroux, J. (1986). A Simple Applicative Language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP'86, pages 13–27. ACM. ↑100

Coquand, T., Huet, G., et al. (1984). The Coq Proof Assistant. `http://coq.inria.fr`. ↑14, ↑15

Curry, H. B. (1934). Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590. ↑32

de Amorim, A. A. (2016). Binding Operators for Nominal Sets. *Electronic Notes in Theoretical Computer Science*, 325:3–27. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII). ↑193

de Bruijn, N. (1991). Telescopic mappings in typed lambda calculus. *Information and Computation*. ↑122

de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392. ↑121, ↑127

Delaware, B., Oliveira, B. C. d. S., and Schrijvers, T. (2013). Meta-Theory à la Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'13, pages 207–218. ACM. ↑24, ↑35, ↑41, ↑68, ↑75, ↑101, ↑202

Filinski, A. (1999). Representing Layered Monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'99, pages 175–188. ACM. ↑103

Filinski, A. (2007). On the relations between monadic semantics. *Theoretical Computer Science*, 375(1):41–75. ↑104

Filinski, A. (2010). Monads in action. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'10, pages 483–494. ACM. ↑104

Frege, G. (1879). *Begriffsschrift: Eine Der Arithmetische Nachgebildete Formelsprache des Reinen Denkens*. L. Nebert. ↑142

Gacek, A. (2008). The Abella Interactive Theorem Prover. In Armando, A., Baumgartner, P., and Dowek, G., editors, *Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer. ↑14, ↑157, ↑190

Gambino, N. and Hyland, M. (2004). Wellfounded Trees and Dependent Polynomial Functors. In Berardi, S., Coppo, M., and Damiani, F., editors, *Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 210–225. Springer. ↑64, ↑188

Gentzen, G. (1935). Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210. ↑7

Gibbons, J. and Hinze, R. (2011). Just Do It: Simple Monadic Equational Reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP'11, pages 2–14. ACM. ↑78, ↑82, ↑106

Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press. ↑153

Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. (1977). Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1):68–95. ↑37

Huet, G. (1997). The Zipper. *Journal of Functional Programming*, 7(5):549–554. ↑184

Huffman, B. (2012). Formal Verification of Monad Transformers. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP'12, pages 15–16. ACM. ↑106

Hughes, J. (2000). Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111. ↑105

Hutton, G. and Fulger, D. (2008). Reasoning About Effects: Seeing the Wood Through the Trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*. ↑105

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450. ↑3

Jansson, P. and Jeuring, J. (1997). PolyP – a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'97, pages 470–482. ACM. ↑63, ↑73, ↑76

Jaskelioff, M. (2011). Monatron: An Extensible Monad Transformer Library. In Scholz, S.-B. and Chitil, O., editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248. Springer. ↑103

Jaskelioff, M., Ghani, N., and Hutton, G. (2011). Modularity and Implementation of Mathematical Operational Semantics. In McBride, C. and Capretta, V., editors, *Proceedings of the Second Workshop on Mathematically Structured Functional Programming,* MSFP'08, volume 229 of *Electronic Notes in Theoretical Computer Science*, pages 75–95. ↑105

Jaskelioff, M. and Rypacek, O. (2012). An Investigation of the Laws of Traversals. In Chapman, J. and Levy, P. B., editors, *Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming,* MSFP'12, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 40–49. Open Publishing Association. ↑64, ↑188

Jones, M. P. and Duponcheel, L. (1993). Composing monads. Research Report YALEU/DCS/RR-1004, Yale University. ↑103

Kaiser, J., Schäfer, S., and Stark, K. (2018). Binder Aware Recursion over Well-scoped De Bruijn Syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 293–306. ACM. ↑191

Kammar, O., Lindley, S., and Oury, N. (2012). Handlers in action. In *The 1st ACM SIGPLAN Workshop on Higher-Order Programming with Effects*, HOPE '12. ↑104, ↑206

Keuchel, S. (2016). Unidb. `https://github.com/skeuchel/unidb-coq`. ↑141, ↑191, ↑211

Keuchel, S. and Jeuring, J. T. (2012). Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming,* WGP '12, pages 57–68. ACM. Copenhagen, Denmark, September 12, 2012. ↑122, ↑191

Keuchel, S. and Schrijvers, T. (2012). Modular monadic reasoning, a (co-) routine. Unpublished. ↑106

Keuchel, S., Weirich, S., and Schrijvers, T. (2016). Needle & Knot: Binder Boilerplate Tied Up. In Thiemann, P., editor, *Proceedings of the 25th European Symposium on Programming,* ESOP'16, volume 9632 of *Lecture Notes in Computer Science*, pages 419–445. Springer. Eindhoven, The Netherlands, April 2–8, 2016. ↑145, ↑153, ↑163, ↑164, ↑196, ↑197

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145. ↑157

Lämmel, R. and Jones, S. P. (2003). Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37. ACM. ↑73

Lee, D. K., Crary, K., and Harper, R. (2007). Towards a Mechanized Metatheory of Standard ML. POPL '07, pages 173–184. ↑190

Lee, G., Oliveira, B. C. d. S., Cho, S., and Yi, K. (2012). GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In Seidl, H., editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 436–455. Springer. ↑75, ↑190, ↑192, ↑196

Leroy, X. (2009). A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363. ↑15

Levy, P. B. (2006). Monads and adjunctions for global exceptions. *Electronic Notes in Theoretical Computer Science*, 158:261–287. ↑104

Liang, S. and Hudak, P. (1996). Modular denotational semantics for compiler construction. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 219–234. Springer-Verlag. ↑106

Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343. ACM. ↑80, ↑103

Licata, D. R. and Harper, R. (2009). A universe of binding and computation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 123–134. ACM. ↑191

Löh, A., Clarke, D., and Jeuring, J. (2003). Dependency-style Generic Haskell. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 141–152. ACM. ↑73

Löh, A. and Magalhães, J. P. (2011). Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 1–12. ACM. ↑73

Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A generic deriving mechanism for haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 37–48. ACM. ↑73

Magalhães, J. P. and Löh, A. (2012). A formal comparison of approaches to datatype-generic programming. In Chapman, J. and Levy, P. B., editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–67. Open Publishing Association. ↑63, ↑74

Malcolm, G. (1990). *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen. ↑37

McBride, C. (2010). Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*. ↑47

Mcbride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13. ↑105

Mendler, N. P. (1987). Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science,* LICS'87, pages 30–36. IEEE Computer Society Press. ↑38

Mendler, N. P. (1991). Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159–172. ↑38

Mitchell, N. and Runciman, C. (2007). Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 49–60. ACM. ↑73

Moggi, E. (1989). An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science. ↑103

Moggi, E., Bell, G., and Jay, C. (1999). Monads, shapely functors and traversals. *Electronic Notes in Theoretical Computer Science*, 29:187–208. {CTCS} '99. ↑64, ↑188

Morris, P. (2007). *Constructing universes for generic programming*. PhD thesis, The University of Nottingham. ↑46, ↑76

Morris, P., Altenkirch, T., and Ghani, N. (2007). Constructing strictly positive families. In *Proceedings of the thirteenth Australasian symposium on Theory of computing - Volume 65*, CATS'07, pages 111–121. Australian Computer Society, Inc. ↑74

Morris, P., Altenkirch, T., and McBride, C. (2006). Exploring the Regular Tree Types. In Fillitre, J.-C., Paulin-Mohring, C., and Werner, B., editors, *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin Heidelberg. ↑63, ↑73, ↑192

Mosses, P. D. (2004). Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 6061(0):195–228. ↑105

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer. ↑14

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University. ↑14

Oliveira, B. C., Hinze, R., and Löh, A. (2006). Extensible and modular generics for the masses. *Trends in Functional Programming*, 7:199–216. ↑73

Oliveira, B. C. d. S., Schrijvers, T., and Cook, W. R. (2010). EffectiveAdvice: disciplined advice with explicit effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 109–120. ACM. ↑78, ↑81, ↑82, ↑106

Pfenning, F. and Elliott, C. (1988). Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208. ACM. ↑121

Pfenning, F. and Paulin-Mohring, C. (1990). Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag. ↑37

Pfenning, F. and Schürmann, C. (1999). Twelf – A Meta-Logical Framework for Deductive Systems. In *CADE-16*. Springer. ↑14, ↑157, ↑190

Pientka, B. and Dunfield, J. (2008). Programming with Proofs and Explicit Contexts. PPDP '08, pages 163–173. ACM. ↑190

Pientka, B. and Dunfield, J. (2010). Beluga: A Framework for Programming and Reasoning with Deductive Systems. In *IJCAR*. Springer. ↑14, ↑157, ↑190

Pierce, B. C. (2002). *Types and Programming Languages*. MIT press. ↑3, ↑52, ↑86

Pitts, A. M. (2003). Nominal logic, A First Order Theory of Names and Binding. *Information and Computation*, 186. ↑121, ↑192

Pitts, A. M., Matthiesen, J., and Derikx, J. (2015). A Dependent Type Theory with Abstractable Names. *Electronic Notes in Theoretical Computer Science*, 312:19–50. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014). ↑142

Plotkin, G. and Pretnar, M. (2009). Handlers of algebraic effects. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer. ↑104, ↑206

Plotkin, G. D. and Power, J. (2002). Notions of computation determine monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 342–356. Springer-Verlag. ↑104

Pollack, R., Sato, M., and Ricciotti, W. (2012). A Canonical Locally Named Representation of Binding. *Journal of Automated Reasoning*, 49(2):185–207. ↑121

Polonowski, E. (2013). Automatically Generated Infrastructure for de Bruijn Syntaxes. In *ITP'13*, volume 7998 of *LNCS*. Springer. ↑190, ↑193

Pottier, F. (2013). dblib. `https://github.com/fpottier/dblib`. Accessed: 2016-07-04. ↑190, ↑191

Pouillard, N. and Pottier, F. (2010). A Fresh Look at Programming with Names and Binders. ICFP'10, pages 217–228. ACM. ↑154

Project, T. T. (2015). The Twelf Wiki. `http://twelf.org/wiki`. Accessed: 2015-10-14. ↑190

Reynolds, J. C. (1983). Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523. ↑106

Sabry, A. and Felleisen, M. (1993). Reasoning about programs in continuation-passing style. *LSC*, 6(3-4). ↑141

Sato, M. and Pollack, R. (2010). External and internal syntax of the λ-calculus. *JSC*, 45(5):598 – 616. ↑121, ↑142

Schäfer, S., Smolka, G., and Tebbi, T. (2015a). Completeness and Decidability of De Bruijn Substitution Algebra in Coq. In *CPP'15*. ACM. ↑141, ↑191

Schäfer, S., Tebbi, T., and Smolka, G. (2015b). Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In Zhang, X. and Urban, C., editors, *ITP'15*, LNAI. Springer. ↑190, ↑191, ↑196

Schrijvers, T. and Oliveira, B. C. d. S. (2010). The Monad Zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven. ↑105

Schwaab, C. and Siek, J. G. (2013). Modular type-safety proofs in Agda. In *Proceedings of the 7th workshop on Programming languages meets program verification*, PLPV'13, pages 3–12. ACM. ↑54, ↑73, ↑74

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strniša, R. (2010). Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20. ↑157, ↑158

Shinwell, M. R., Pitts, A. M., and Gabbay, M. J. (2003). FreshML: Programming with Binders Made Simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 263–274. ACM. ↑158

Stansifer, P. and Wand, M. (2014). Romeo: A System for More Flexible Binding-safe Programming. In *ICFP'14*, pages 53–65. ACM. ↑154, ↑157, ↑158

Steele, Jr., G. L. (1994). Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 472–492. ACM. ↑103

Stump, A. and Tan, L.-Y. (2005). The Algebra of Equality Proofs. In *International Conference on Rewriting Techniques and Applications*, pages 469–483. Springer. ↑189

Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. ↑27, ↑29, ↑75, ↑105

Tate, R. (2017). `https://dev.to/rosstate/java-is-unsound-the-industry-perspective`. Accessed: 2018-05-15. ↑3

Urban, C., Berghofer, S., and Norrish, M. (2007). Barendregt's Variable Convention in Rule Inductions. In Pfenning, F., editor, *Automated Deduction – CADE-21*, pages 35–50. Springer. ↑121

Urban, C. and Tasson, C. (2005). Nominal Techniques in Isabelle/HOL. In *CADE-20*, volume 3632 of *LNCS*. Springer. ↑158, ↑190, ↑192

Uustalu, T. and Vene, V. (2000). Coding recursion à la mendler. In *Proceedings 2nd Workshop on Generic Programming, WGP '00*, pages 69–85. ↑38

Van Noort, T., Yakushev, A. R., Holdermans, S., Jeuring, J., Heeren, B., and Magalhães, J. P. (2010). A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(3-4):375–413. ↑17, ↑63

Verbruggen, W., de Vries, E., and Hughes, A. (2008). Polytypic programming in Coq. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 49–60. ACM. ↑73

Voigtländer, J. (2009). Free theorems involving type constructor classes: functional pearl. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 173–184. ACM. ↑106

Vouillon, J. (2012). A Solution to the PoplMark Challenge Based on de Bruijn Indices. *Journal of Automated Reasoning*, 49(3). ↑156, ↑195

Wadler, P. (1989). Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359. ACM. ↑106

Wadler, P. (1992). Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*. ↑103

Wadler, P. (1998). The Expression Problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. Accessed: 2017-05-10. ↑23, ↑26

Wadler, P. (2015). Propositions as types. *Commun. ACM*, 58(12):75–84. ↑33

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2004). A concurrent logical framework: The propositional fragment. In *TYPES*, volume 3085 of *LNCS*. Springer. ↑141

Weirich, S. (2006). RepLib: A Library for Derivable Type Classes. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 1–12. ACM. ↑192

Wright, A. and Felleisen, M. (1994). A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1). ↑112

Yakushev, A. R., Holdermans, S., Löh, A., and Jeuring, J. (2009). Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 233–244. ACM. ↑63, ↑73

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM. ↑15

Zhao, J., Zhang, Q., and Zdancewic, S. (2010). Relational Parametricity for a Polymorphic Linear Lambda Calculus. In Ueda, K., editor, *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, pages 344–359. Springer Berlin Heidelberg. ↑15