Optimal Interaction Strategies using Reflection in LTS: A Demonstration

Thomas Cleenewerck [1]

Programming Technology Lab
Vrije Universiteit Brussel
Brussel, Belgium

Abstract

The semantics of languages, including the semantic rules which valid programs must obey, erect many interactions among language constructs. When implementing the interactions, the resulting coupling among constructs cripple future evolutions. Effectively reducing this coupling requires a large arsenal of implementation techniques, which we refer to as interaction strategies. These range from off the shelf strategies, to customized strategies up to tailored and specifically designed strategies. In this demonstration, we show how the reflective layer in the Linglet Transformation System (LTS) supports the existing arsenal of strategies, how they can be adapted and how new strategies can be defined. Using LTS, developers can apply the optimal strategy to establish the interactions among the different language constructs with a minimum amount of coupling.

Key words: language, transformation, reflection, modularization

1 Introduction

One of the major driving forces in language design is the increase of abstraction level. This continuing endeavor leads to new language constructs that better reflect how programmers want to encode programs. As a result, language implementations need to be extended with the syntactical and semantical definitions of those language constructs. To facilitate this evolution process we strive to structure the implementation of languages according to their language constructs. In such implementations, each construct is implemented by a single module. The premise is that evolving such implementations in terms of language constructs simply boils down to adding, removing or changing the modules which correspond to the language constructs that are involved. Unfortunately, the semantics of languages – including the semantic rules which

[1] Email: tcleenew@vub.ac.be

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

valid programs must obey, and the optimizations – are defined as interactions among several language constructs. Examples of such interactions are naming analysis, type checking, symbol tables, weaving of advices, etc. So when attempting to modularize language implementations according to their language constructs, the semantic interactions crosscut this modularization.

In most language development techniques (LDTs) the crosscutting interactions can be specified separately from the rest of the semantics of language constructs. For example, with rewrite rule systems separate rewrite phases can be defined to handle the interactions; with attribute grammars interactions can be defined in separate aspects. However, mere separation does not suffice to fulfill the evolution premise as the coupling among the language constructs because of the interactions still has to be resolved. It is this coupling that cripples future evolutions.

Most LDTs provide their own particular and specific implementation mechanism to reduce the coupling among the language constructs. We call them interaction strategies. Examples of such interaction strategies are among others: attribute propagation rules [4], forwarding [9], structure-shy queries [8], traversals [7] and symbol tables. Each of them is designed to support a particular interaction pattern among constructs. They have been carefully developed and have proven their strengths and weaknesses over quite some time. An analysis of the current strategies and the need to improve them is given in [1]. Alas, interaction strategies are fixed and embedded in contemporary LDTs. This prohibits language developers to use an optimal interaction strategy for decoupling language constructs.

In this demonstration, we show that coupling can be more effectively reduced if a wide range of interaction strategies is made available to the language developer. We start the demo by showing the different pros and cons of some of the current off the shelf strategies. In the second step, we suggest adaptations to existing strategies to be able to cope with some of the deficiencies to further minimize the coupling among language constructs. In a last step, entirely new strategies are designed and implemented that will allow us to reduce coupling in cases where the interaction pattern of current strategies does not fit at all.

The environment we use to demonstrate the above is called the Linglet Transformation System (LTS) [1]. LTS preserves the modularization of the semantics of language constructs in the presence of crosscutting interactions by offering a reflective layer to implement interaction strategies. It invites designers to experiment with various strategies so as to face new coupling problems and minimize coupling.

2 Linglet Transformation System

The Linglet Transformation System is a development technique for the implementation of languages. The system is divided into two layers: a kernel

for the definition of the language constructs and their composition into a language, and a reflective layer to establish the interactions among the language constructs.

2.1 Kernel

LTS strictly modularizes language implementations into modules called linglets. Each linglet defines the syntax and the semantics of a single language construct in isolation from any other language construct. Informally, this means that both its syntax and semantics must not depend on the existence of other language constructs ranging from a direct reference to any implementation decision that is imposed by another construct. A formal description can be found in [1].

LTS is a prototype-based object-oriented language where each linglet is a prototypical object that defines a language construct. The syntax and semantics of a linglet are its behavior. They are thus implemented by methods.

The syntax of a linglet is defined in a syntactical method using a higher order grammar. Simply put, in a higher order grammar the non-terminals are replaced by syntactical parameters, e.g. an `IF` linglet contains three syntactical parameters namely `condition`, `consequent` and `alternate`. These can in turn be bound to other linglets, when the linglets are composed together to define a language, e.g. the `condition` syntactical parameter to a `BooleanExpression` linglet and the `consequent` and `alternate` a `Statement` linglet. Hence, the syntax of a linglet does not refer to another language construct and is thus defined in isolation.

A program is represented in LTS with an abstract syntax tree (AST). Upon parsing a linglet, an AST node is created by instantiating the linglet. Its datamembers or parts correspond to the syntactical parameters of a linglet e.g. an `IF` AST node has three parts namely `condition`, `consequent` and `alternate`.

The semantics of a linglet is defined in a method that interprets the linglet or that translates the linglet to a semantically equivalent code fragment which is written in some target language. The semantics uses the parts of a linglet, but can also request additional information. As the linglet is isolated, this information must be declared as an abstract method. When using the linglet in a language, the language designer has to provide this information. Linglets can also produce multiple results or multiple target language program fragments. Again, to ensure the isolation of the linget, the handling of these fragments (integration in various parts of the target program) is also done when the linglet is used in a language.

A language and its implementation are respectively defined and constructed by combining the necessary linglets. The language is defined by the bindings of the syntactical parameters. The language implementation is defined by providing the necessary information to the linglets and by handling multiple

results [2] .

2.2 Reflective Layer

The semantic interactions among constructs may stretch from neighboring linglets [3] to distant linglets. Accessing them by traversing the parent or child nodes in the AST, requires many small interactions. This erects many dependencies among linglets, which cripple future evolutions of the language. For this, interactions among language constructs can be implemented based on so called *interaction strategies*. Interaction strategies implement mechanisms that are used amongst others to exchange information or integrate multiple results or target language program fragments.

Interaction strategies are defined in a reflective layer which is defined on top of the kernel. Because of this, the linglets do not have to be changed and their modularity is guaranteed. Consider for example the handling of multiple results which need to be integrated in various places of the target AST. Without a reflective layer, the behavior of other linglets would have to be changed in order to process multiple results and to handle them appropriately: either by integrating them into their own results or by propagating them to other linglets. By defining the strategy, linglets do not have to be changed and their modularity is guaranteed.

A computational reflective layer aligns nicely with the activities of interaction strategies. There are two types of computational reflection: structural and behavioral. Both are required in order to implement strategies as they inspect and intercede the linglets and change their semantics.

Structural reflection for LTS grants us the ability to inspect and manipulate the relationships among instantiated linglets, inspect the linglet's syntactical parameters, inspect the behavior offered by a linglet and manipulate the behavior of a linglet. For example: structure-shy queries reason about the relationships of the instantiated linglets. Symbol tables add additional behavior to the linglet for accessing values.

Behavioral reflection for LTS grants us the ability to change how linglets respond to requests, to intercept how they are connected, how they construct target program fragments, etc. For example: attribute copy- and propagation rules determine whether a linglet can respond to a request. If the linglet cannot respond to the request then it is propagated to a neighbor. Symbol tables propagate its mappings throughout the execution of the transformation.

The reflective layer in LTS is organized as a metaobject protocol (MOP) such that it can be extended in an object-oriented style [3,2,5]. Object-oriented

[2] In general, language implementations have more responsibilities. They must also resolve semantic and syntactical compositionality conflicts and ensure coherence and cooperation.
[3] These are the linglets that are bound its syntactical parameters

concepts like dynamic dispatch, inheritance and information hiding are used at the meta-level to provide abstractions that hide implementation details of LTS while at the same time ensuring its extensibility. Moreover the collaborations among linglets in a interaction strategy naturally map to the concept of protocols and subprotocols in a MOP, resulting in an elegant system.

3 Demonstration

We demonstrate how LTS provides a wide range of interaction strategies to the language developer in order to more effectively reduce the coupling among language constructs. The case which we use throughout the demonstration is the implementation of the tuple query language in terms of the structured query language. We start by introducing the kernel of LTS, we then cover the following topics regarding the reflective layer in our demonstration:

Existing Strategies

The reflective layer in LTS supports a wide range of existing strategies. As a first introduction to the reflective layer, we show how to extend the LTS system with a number of well known strategies such as structure-shy queries, traversals, synthesized and inherited attributes, attribute copy rules and symbol tables. By implementing them in LTS, the strategies share a common implementation model. This allows us to easily study their differences. We will take advantage of this fact by showing the different pros and cons of the implemented, current, off the shelf strategies.

Adaptations to Existing Strategies

In the second step, we suggest adaptations to existing strategies to be able to cope with some of their deficiencies to further minimize the coupling among language constructs. We start by demonstrating the basic techniques offered by LTS to adapt strategies. In the examples, we also illustrate the main sources of inspiration to adapt the existing strategies namely cross-fertilization among LDTs by implementing the ordering relation of HyperJ [6], adaptation to a suitable abstraction level by extending the ordering relation of HyperJ, resolving conflicts among strategies by tackling the conflict between attribute forwarding and attribute copy rules, and generalizing strategies by treating forwarding as a special case.

New Strategies

In a last step, we use a gradually more complex scenario where we show that an existing strategy has to be changed and finally can become inadequate. We start with structure-shy queries, subsequently extend the strategy and finally replace it by a entirely new strategy called INR. The ability to design

and implement entirely new strategies allow us to reduce coupling in cases where the interaction pattern of current strategies does not fit at all.

4 Conclusion

The semantics of languages crosscuts the semantics of individual language constructs. Contemporary language development techniques tackle the crosscutting itself by separating the interactions from the constructs, but offer only limited support to reduce the coupling among the constructs. They do this in the form of embedded and fixed interaction strategies. In this demonstration, we argue that to further reduce the coupling, language developers should have a large arsenal of interaction strategies at their disposal. The argument is demonstrated by showing that a wide range of existing strategies have different tradeoffs, that they need to be adapted and that new strategies are necessary. The implementation of this wide range of strategies is conducted in the linglet transformation system (LTS). LTS offers a reflective layer on top of its kernel for implementing strategies. As such, the transformation system can be extended with an optimal strategy to effectively minimize the coupling among language constructs.

References

[1] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach.* PhD thesis, Vrije Universiteit Brussel, 2007.

[2] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[3] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[4] K. B. Oege de Moor and S. D. Swierstra. First Class Attribute Grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications.

[5] P. Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, 1994.

[6] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* IBM Research, 2000.

[7] M. G. J. Van Den Brand, P. Klint, and J. J. Vinju. Term Rewriting with Traversal Functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[8] R. Whitmer. Document Object Model (DOM) Level 3 XPath Specification W3C, March 2002.

[9] E. V. Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.