Towards a Product Line of Interpreters: An Experiment with Textbook Languages

Thomas Cleenwerck, Rodolfo Toledo

Index Terms—D.3.4.e Programming Languages, Processors, Interpreters

1 Introduction

As software is subjected to a continuing rate of evolution, the programming languages that were used to construct it evolve as well. This is not only apparent from a historical perspective, where we see that all mainstream languages continue to evolve. It is even more so apparent in our continuous effort to construct languages which are designed with a particular application domain in mind. Examples of this range from computational domains like object-orientation, aspects, over hardware domains like distribution or parallelism, to end-user application domains like business process engineering.

In this paper we focus on the evolution of the semantics of a programming language. Ideally, languages should be extensible using modular extensions. This implies that the impact of change to an existing language as well as the changes to the already applied extensions should be minimal.

A wealth of techniques has been proposed and investigated to facilitate language implementations. However, we observe that in these techniques semantics are still intertwined with a particular implementation strategy that is shared among many if not all the language features. The experiments in this paper show that the semantics of features are often too coarse grained. This cripples the ability to easily combine them, and thus hinders evolution. The semantics of features that contain boilerplate specifications can relatively easily made modular. However, dependencies on a shared interaction among different language features are harder to tackle and require specific abstractions. Lastly, entirely new mechanisms are needed to capture the subtile changes in semantic specifications in order to yield a coherent overall language semantics.

In order to better support the evolution of languages, we first require a better understanding of how languages evolve. More precisely, in the first part of the paper, we investigate how the different features of languages affect one another. We turn a series of language evolutions into a product family [PBvdL] by studying their commonalities and variabilities. We then analyze the current implementations from this point of view. In the second part, we determine how we can decouple the shared implementation strategy from the language features and ultimately present some indications as to how we can specify the semantics as modular composable extensions by increasing the abstraction level of semantical specifications. We start from a straightforward scheme implementation as presented in [Kri96] and improve it with the implementation techniques found in the Linglet Transformation System (LTS) [Cle07].

- *T. Cleenewerck works at the PROG Lab of Computer Science Department, Vrije Universiteit Brussel, Belgium*
  E-mail: tcleenew@vub.ac.be
- *R. Toledo works at the PLEID Lab of Computer Science Department (DCC), University of Chile, Chile*
  E-mail: rtoledo@dcc.uchile.cl

2 Motivating Problem: Evolution Graph of Interpreters

In the text book by Krishnamurthi [Kri96], programming languages are thought by means of interpretation and application. Programming language concepts are gradually introduced to students by incrementally adding new language features. The author starts with a language for simple arithmetics called AE. He then creates the language WAE where he adds the `with` construct with substitution semantics. He further evolves the WAE language with `first-order functions` with substitution and then later with a store to attain the languages called F1WAE and F1WAE (ds[1]) respectively. This process continues with language features like `functions`, `conditional expressions`, `lazy evaluation` etc.

The language evolutions follow a particular pedagogical scenario. Because of this, little or no attention is paid on separating the impact on the interpreter when changing from one version to another. In other words, language evolution is basically handled by copy-pasting a previous version of an interpreter and subsequently modifying it. Hence, the various concepts are not clearly *separately* defined. Changes to previous semantics are easily overlooked, especially the subtle ones. There are several drawbacks of this approach.

1. The term "ds" stands for data store or just store.

- First, concepts are harder to understand and reason about. For example, it is hard to understand what the semantics of a function application are, irrespective of using substitution or a store.
- Second, it is harder to understand the impact of a concept on other language features. For example, it is hard to understand the impact of adding the feature `with` (with substitution-based semantics) on a language implementation.
- Third, implementation techniques are not made fully explicit and cannot thus be reused as such. For example, it is not possible to reuse the same `store` for first order functions in a language with higher order functions or with boxes.
- Fourth, experimenting with combinations of language features that deviate from the standard pedagogical scenario is complicated, as such implementations require to cut and paste from different language versions. For example, it is not possible to easily construct a language with a store but without first order functions.

In order to accommodate these needs, we analyzed the commonalities and variabilities of this series of language evolutions so as to turn them into a product family of languages.

3 Commonalities & Variabilities

As the language evolutions of the interpreters are steered by a pedagogical scenario, features are piled up so as to gradually expose students to more complicated semantics and features. Moreover, the scenario only implicitly states which features depend on the existence of other features.

In order to attain a product family, we have conducted a communality and variability analysis using FODA [KCH+90], [Cza98], augmented with some constraints to capture dependencies among features.

The resulting model reveals that many more combinations can be explored. For example, each language feature is hierarchically subdivided into subfeatures. The feature `identifier` has three subfeatures: referencing existing identifiers, defining new identifiers and changing the value associated to identifiers. A specific language is the result of selecting the features of interest. For example, the language called F1WAE can be defined by selecting the following features: all `arithmetic` expressions, `identifier references and definitions` and `function application`.

The model also exposes all of the choices that have to be made when implementing a language. Languages are not solely determined by their features, also the implementation techniques that are used to implement the features and their interactions are made explicit in the model. Amongst others, these are `threading` and `substituion`. Threading transports bindings from language features that define or set values (e.g. identifier definitions or updates) to language features that refer to the bindings (e.g. identifier references), whereas substitution replaces the bindings in the language features.

In addition, the model captures explicitly the dependencies among the existence of features. The hierarchical nature of the model already entails some dependencies e.g. `identifiers`, when chosen, must at least be referentiable. Using constraints, dependencies among features can be defined which cannot be captured hierarchically e.g. arithmetic operators must fit the available datatypes.

4 Coarse-grained semantics

When analyzing the straightforward implementations of our series of language interpreters in scheme we found that language features are too coarse grained. The result is that language evolutions cannot simply reuse and extend the semantics of previous versions, but have a significant impact. More precisely, we observe that the semantics suffer from reuse of boilerplate code or contains dependencies on shared interactions and on other language features.

4.1 Boilerplate code

The semantics encoded in the interpreters suffer from quite a lot of boilerplate code. Consider for example the implementation of `with` using substitution semantics. The `with` feature introduces an identifier which names, or identifies, an expression and allows to use this name in a larger computation. We refer to the later computation as a sub expression. Upon evaluation, the semantics of `with` substitute the expression bound to the identifier within its sub expression. Substitution has to traverse the whole sub expression in order to find all occurrences of a identifier. It thus impacts all the other language features which can be used as (a part of) an expression. However, only a couple language features are worthwhile to consider, e.g. `identifier reference` and `with` itself. In case of constructs such as `addition`, substitution passes through, and even in case of constructs such as a `number` nothing has to happen at all.

4.2 Dependencies on shared interactions

Semantics of features encoded in the interpreters are polluted with specifications that depend on a shared interaction among different language features. Consider for example an `identifier reference`. Its implementation is polluted by the kind of semantics that are used by other language features. In a substitution-based semantics, identifier references are substituted away. Hence, the semantics of a (unsubstituted) identifier reference produce an error. In case the semantics are defined with a store, references are looked up in a given store. Despite these differences, in fact, both implementations return the value which is associated to identifiers. In the former case, it is an error and in the latter case a value from the store.

We encounter the same problem in function applications. Irrespective of a substitution or store semantics, a function application in essence binds the parameters of a function to its arguments, evaluates a function, and removes the binding. However, the semantics of function applications when specified in substitution, substitute the formal parameter in its body whereas with a store, it extends the store, creates a new binding and afterwards restores the store again.

4.3 Dependencies on other language features

Semantics encoded in the interpreters also depend on other language features in order to yield a coherent overall language semantics. In a lazy interpreter, for example, there are some points where the implementation of a lazy language forces an expression to reduce to a value (also known as the strictness points of the language). These strictness points occur in many other language features e.g. upon the evaluation of an addition, the strictness point ensures that actual values are produced so that the interpreter can compute their sum. Hence, when making languages lazy the original semantics cannot be reused, but have to be carefully examined and changed.

5 Towards modularly composable extensions by increasing the abstraction level

In this section, we analyze how we can decouple the shared implementation strategy from the language features. We change the scheme implementation using state of the art language development techniques and postulate how we can further improve the implementation in order to construct the language evolutions as modular extensions.

For our experiments we use LTS [Cle07]. LTS serves as a experimental environment as it combines the strengths of a large amount of language development techniques and cultivates (and to some extent enforces) a discipline to modularize the semantics of languages. First, LTS strictly modularizes the syntax and semantics of each language construct in a language module, called a linglet. In LTS, languages are built by composing linglets. As a result, language extensions are defined modularly by adding and recomposing linglets. Second, LTS features the unique ability of being customizable. This allows developers to adopt the most optimal implementation for separating the different language features. As such, developers can use advanced interaction strategies and composition mechanisms in order to establish the semantics of a language, while ensuring its modular construction.

5.1 Abstracting from Boilerplate code

Let us revisit the language extension by the `with` feature using substitution semantics. The boilerplate code, which does not contribute to the semantics, can be removed in several steps by several techniques.

In a first step, we share common substitution behavior among different language features. For example, the substitution semantics of all binary operations can be shared among features such as `addition` and `substraction` i.e. a new AST node is created where the substitution is applied to the right and left. Likewise, the substitution semantics of all terminals can be shared among features such as `identifiers` and `integers` i.e. the current AST node is returned.

In a second step, we abstract from the substitution semantics of features such as terminals and binary operations as no specific semantics have to be executed. In fact, the substitution just "passes through". This can be abstracted by traversing the AST. The traversal itself is defined on a meta level in order to abstract from the specific AST nodes. With this technique all boilerplate substitution semantics can then be omitted.

In a last step, we extract a reusable mechanism from the above traversal which we call the propagation interaction strategy[2] (PIS). We do this by parameterizing the function which is propagated. This mechanism can thus propagate any function call top-down on an AST.

We end up with an implementation of substitution where only the language constructs are involved which have particular semantics, and with an explicit mechanism to implement it.

5.2 Dependencies on shared interactions

Sometimes the semantics of language constructs are too dependent on an interaction that involves other language constructs. Recall in our example that the semantics of a function application either have to initiate the substitution or either has to change the store. In turn, this choice completely changes the semantics of an identifier reference.

What is required here is a way to abstract from the actual way the semantics are implemented. In case of a identifier reference, the semantics need to retrieve its value. LTS's modularization mechanism provides an abstraction to deal with missing information. Consider the following three slightly different implementations. The semantics of an identifier reference can:

- in case of substitution semantics, produce an error indicating that a value has not been found;
- alternatively, also indicate that an error value is necessary by parameterizing their semantics with a lambda;
- in case of store semantics, simply assume that the value is available (abstract from its definition site) and lookup the value in the current AST node.

2. The term interaction strategies refers the fact the traversal deals with multiple language features

Upon composition of identifier references in a language, one does not need to distinguish between those three implementation techniques. LTS provides an abstraction from these three implementations and can uniformly handle them, because in essence in all cases a value is requested that is not available.

In some cases semantic dependencies have to be explicitly dealt with. Let us revisit the semantics of a function application. The implementation of its semantics is quite different depending on whether substitution or a store is used. Following the strict discipline of LTS to modularize the semantics, a function application could adopt an operational semantic style using the abstract notion of a binding. Hence, the semantics consists of these three steps:

1) extend the state of evaluation with a new binding,
2) execute the semantics of the function,
3) revert the state of evaluation by cancelling the binding.

This formulation allows developers to reuse the semantics of function application regardless of a substitution style or store style. In case of a substitution style:

1) change the state of evaluation to a new program where the identifier is substituted
2) execute the semantics of the function,
3) do nothing, as the state of evaluation does not have to be reverted

In case of a store:

1) change the state of evaluation to an extended new environment containing the identifier
2) execute the semantics of the function,
3) revert the state of evaluation to the environment where the new binding is removed

To conclude, semantic dependencies on shared interactions can be avoided by raising the abstraction level of the implementation of the semantics. As we have demonstrated, for some dependencies LTS provides some build in semantics to implicitly abstract from these dependencies, for other dependencies explicit care is needed. Future work has to determine whether new abstractions can be implicitly supported and how.

5.3 Dependencies on other language features

The case where semantics also depend on other language features to yield a coherent overall language semantics is the most delicate task. The reason for this is that subtle differences occur deep inside the semantics of a feature and possibly many features. The challenging part is to specify the correct locations and the changes in semantics without violating the modularity of language features. Note that the latter is not simply a restriction but rather a fundamental requirement to avoid brittle language extensions.

In the example given in the previous section, lazy evaluation impacts quite a lot on other language features. More so, the places where strictness applies are difficult to assess. Our goal is to be able to declaratively specify the locations. For the example of strictness, strictness has to be applied whenever the interpreter executes a function of the host language. However, this definition only covers a subset of all the places in the interpreter. To complement this definition, we are increasing the abstraction level of the semantics so as to better expose the assumptions taken. In case of strictness, we could require developers to tag the operations executed by the interpreter. With these tags we could indeed declaratively specify all locations where strictness applies.

5.4 Discussion

LTS's strict modularization and advanced interaction and composition mechanisms brought us quite close to a modular implementation of the considered language evolutions. Boilerplate code could be removed and dependencies on shared interactions could be avoided to some degree. However, dependencies on other language features posed a greater challenge. In fact, concerning the two kinds of dependencies, we observed that in our experiments that the granularity of the semantical specifications should be decreased. Simply accessing and intervening in the semantics of other language features would violate their modularity. By consequence, we would fail in our attempt to modularize the interpreter extensions. So, in order to ensure the modularity, the decrease in granularity should be accompanied by a raise in the abstraction level of the semantical specifications.

6 Conclusion

In our continuing effort to improve programming languages so as to better suit the need of developers, languages continuously need to evolve. In this paper, we focus on the ability to evolve the semantics of languages. An analysis of the changes of a wide range of interpreters showed that changes from one language version to the next often has a significant impact on the semantics of the original language. We rewrote the interpreters using state of the art language development techniques involving modular language constructs, reflective interpreter extensions, and complex interaction and composition techniques. A catalog of changes shows that only parts of the extensions could be modularized. This is due to the coarse-grained semantical descriptions using two low-level semantical abstractions. This paper presents some indications as to how we can resolve the situation in order to effectively specify the semantics as modularly composable extensions.

Acknowledgments

The authors would like to thank Éric Tanter for his insights in the problem context of this work and for the fruitful discussions on the topic. They also want to thank Johan Fabry for his comments on this work.

References

[Cle07] Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, 2007.

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, 1998.

[KCH$^+$90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Kri96] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Computer Science Department, Brown University, Providence, MA, USA, 1996.

[PBvdL] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques., publisher= Springer, year=2005*.