Towards a Product Line of Interpreters: An Experiment with Small Functional Languages

Thomas Cleenewerck, Rodolfo Toledo, Éric Tanter

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 be 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 subtle changes in semantic specifications in order to yield a coherent overall language semantics.

- *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
- *É. Tanter works at the PLEID Lab of Computer Science Department (DCC), University of Chile, Chile*
  E-mail: etanter@dcc.uchile.cl

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 [PBvdL05] 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 shared implementation strategies 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 implementation in Scheme as presented in [Kri97] and improve it with the implementation techniques found in the Linglet Transformation System (LTS) [Cle07].

2 Motivating Problem: Evolution Graph of Interpreters

In the text book by Krishnamurthi [Kri97], programming languages are taught 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 consisting of addition and substraction that operate on numbers. He then creates the language WAE where he adds the with construct (also known as let) with explicit[1] semantics. He further evolves the WAE language with first-order functions with explicit and then later with a deferred semantics[2] to attain the languages called F1WAE and F1WAE (env[3]) respectively. This process continues with language features such as first class functions, conditional expressions, lazy evaluation, mutable identifier values.

The language evolutions follow a particular pedagogical scenario. Because of this, 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

1. Semantics is expressed using substitution.
2. Semantics is expressed using an environment
3. The term “env” stands for environment.

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.

- 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 explicit or deferred semantics.
- 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 explicit 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 mechanism that threads an environment used for first order functions to thread a store in a language with mutable identifier values.
- 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 variant of WAE that uses an environment.

In order to accommodate these needs, we analyzed the commonalities and variabilities for a couple of language evolutions 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. 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 mutating 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 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 substitution. Threading transports bindings from language features that define or set values (e.g. identifier definitions or identifier value mutations) to language features that refer to the bindings (e.g. identifier references), whereas substitution replaces the bindings in expressions of these language features.

In addition, the model captures explicitly the dependencies among the 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 be defined with respect to 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 contain 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 explicit semantics in the WAE language. with 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 body. Upon evaluation, the semantics of with substitute the identifier by the expression bound to the identifier within the body of the with. Substitution has to traverse the whole body in order to find all occurrences of a identifier. Hence, for each language feature which can be used as (a part of) an expression, code has to be written. However, only a couple language features are worthwhile to consider, the rest is merely boilerplate code. In the WAE language these are 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 of a WAE language using explicit and deferred semantics. Its implementation is polluted by the kind of semantics that are used by other language features. In an explicit semantics, identifier references are substituted away. Hence, the semantics of a (unsubstituted) identifier reference produce an error. In case of deferred semantics (using an environment),

references are looked up in a given environment. 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 environment.

We encounter the same problem in function applications. Irrespective of an explicit or deferred semantics, a function application in essence binds the parameters of a function to its arguments, evaluates the body of a function, and removes the created bindings. However, the explicit semantics of function application substitute the formal parameter in the body of the function whereas in the deferred semantics of function application extend the environment, create a new binding and afterwards restore the environment 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 have to be specified in the semantics of 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 an 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 such that developers can adopt the most optimal implementation for separating the different language features. This enables developers to 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 explicit 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 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 of terminals can be shared among features such as numbers i.e. the current AST node is returned.

In a second step, we omit substitution for features such as terminals and binary operations as no specific semantics have to be executed. We defined a separate module that offers a mechanism that is capable of propagating the substitution through an expression.

We end up with an implementation of explicit semantics 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 environment. 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 an 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 of an identifier reference in the F1WAE languages. The semantics of an identifier reference can:

- in case of explicit semantics, produce an error indicating that the identifier is not bound;
- alternatively, also indicate that an error value is necessary by parameterizing its semantics with a lambda that provides such a value;
- in case of deferred semantics, simply assume that the value is available (abstract from its definition site) and lookup the value.

Upon composition of identifier references in a language, one does not need to distinguish between those three implementation variants. 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 an environment is used. Following the strict discipline of LTS to modularize the semantics, a function application in the F1WAE language could adopt an operational semantic style using the abstract notion of a binding. Hence, the semantics consists of these three steps:

1) extend the evaluation state with a new binding,
2) interpret the body,
3) revert the evaluation state by cancelling the binding.

This formulation allows developers to reuse the semantics of function application regardless of explicit or deferred semantics. In case of explicit semantics:

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

In case of deferred semantics:

1) change the evaluation state to an extended new environment containing the identifier,
2) interpret the body,
3) revert the evaluation state 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 built 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 twofold: (a) to specify the correct locations and (b) to specify 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 [CD08], instead of depending on low-level implementation details by relying on mere syntactical patterns [GB03]. For the example of strictness, strictness has to be applied whenever the interpreter executes a function of the host language[4]. 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 away from low-level implementation details 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

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 in our experiments that the granularity of semantic specifications should be decreased. This means that we require smaller units of semantic behavior for modularizing the interactions and dependencies on other language features. The question is how much details must be exposed, without compromising the modularity of the language implementation. We therefore opt to approach this problem in a bottom up fashion and commence our search for a suitable level of granularity by raising the abstraction level of semantic 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 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 Johan Fabry for his comments on this work.

4. The host language is the language that executes the interpreter.

REFERENCES

[CD08] Thomas Cleenewerck and Theo D'Hondt. Modularizing Invasive Aspect Languages. In *DSAL '08: Proceedings of the 3rd Workshop on Domain-specific Aspect Languages*, New York, NY, USA, 2008. ACM.

[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.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[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.

[Kri97] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Computer Science Department, Brown University, Providence, USA, 1997.

[PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.