Verifiable composition of language extensions

Ted Kaminski

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
tedinski@cs.umn.edu

Abstract. Domain-specific languages offer a variety of advantages, but their implementation techniques have disadvantages that sometimes prevent their use in practice. *Language extension* offers a potential solution to some of these problems, but remains essentially unused in practice. It is our contention that the main obstacle to adoption is the lack of any assurance that the compiler composed of multiple independent language extensions will work without the need for additional modifications, or at all. We propose to solve this problem by requiring extensions to independently pass a composition test that will ensure that any such extensions can be safely composed without "glue code," and we propose to demonstrate that interesting extensions are still possible that satisfy such a test.

1 Introduction

Domain-specific languages (DSLs) come with a variety of reasonably well-known advantages and disadvantages [3]. Some of these disadvantages do not seem to be inherent to DSLs in general, but are a consequence of the way they are implemented. In particular, many implementation techniques lack or poorly support *composition*, meaning multiple DSLs cannot easily be used together to solve a problem.

To be more precise about what we mean by language composition, we will use some of the classification and notation of Erdweg, Giarrusso, and Rendel [5]. The notation $H \triangleleft E$ represents a *host language* H composed with a *language extension* E, specifically crafted for H. Another composition operator $L_1 \uplus_g L_2$ denotes the composition of two distinct languages with "glue code" g. To permit only the $\triangleleft$ form of language composition ("language extension") is not sufficient. With H, $H \triangleleft E_1$, and $H \triangleleft E_2$, we are left with no option for composing all three, without modifying one of the extensions to have the form $(H \triangleleft E_1) \triangleleft E_2$ (or vice versa.) However, the $\uplus_g$ form of language composition ("language unification") is also insufficient for our purposes. The problem with this form of composition is that the "glue code" g necessary to perform this composition is essentially an admission that the composition is broken and must be repaired. (Though it is still interesting that the composition *can* be repaired.)

What we seek is a composition method $L_1 \uplus_\emptyset L_2$, that is, language unification without needing any glue code ($g = \emptyset$.) This may seem impossible in general, but there is hope in special cases, such as when both languages are extensions to a common host: $(H \triangleleft E_1) \uplus_\emptyset (H \triangleleft E_2)$. Here we are tasked with resolving only conflicts between E_1 and E_2, while the host language H is shared. We will say that a DSL implementation technique supports *composable language extension* if it is capable of composition of the form $H \triangleleft (E_1 \uplus_\emptyset E_2)$. We further require that the technique provides some assurance that the resulting composed language will work as intended, and is not simply broken.

The goal of this work is to build a DSL implementation tool and demonstrate that it satisfies the following criteria:

- Supports composable language extension, as defined above.
- Permits introduction of new syntax.
- Permits introduction of new static analysis on existing syntax.
- Capable of generating good, domain-specific error messages.
- Capable of complex translation, such as domain-specific optimizations.

In Section 2 we provide some background on the tools we will be making use of in pursuit of this goal. In Section 2.1 we survey some of the other tools for implementing domain-specific languages. In Section 3 we propose the work we plan for this thesis. In Section 3.1 we outline work beyond the scope of this thesis.

2 Background

The first major obstacle to supporting composable language extension is to allow composition of syntax extensions. Although context-free grammars are easily composed, the resulting composition may no longer be deterministic, or otherwise amenable to parser generation. Copper [15, 20] is an LR(1) parser generator that supports syntax composition of the form $H \triangleleft (E_1 \uplus_\emptyset E_2)$ so long as each $H \triangleleft E$ individually satisfy some conditions of its *modular determinism analysis*. Assuming we require extensions to satisfy this analysis, Copper offers one solution to the syntax side of the problem of supporting composable language extension.

Attribute grammars [13] are a formalism for describing computations over trees. Trees formed from an underlying context-free grammar are attributed with synthesized and inherited attributes, allowing information to flow, respectively, up and down the tree. Each production in the grammar specifies equations that define the synthesized attributes on its corresponding nodes in the tree, as well as the inherited attributes on the children of those nodes. These equations defining the value of an attribute on a node may depend on the values of other attributes on itself and its children. Attribute grammars trivially support both the "language extension" and "language unification" modes of language composition, by simply aggregating declarations of nonterminals, productions, attributes, and semantic equations.

There is a natural conflict between introducing new syntax and static analysis, referred to as the "expression problem[1]." Although normally formulated in

[1] `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

terms of data types, it applies equally well to abstract syntax trees, and thus has consequences for language extension. If one language extension introduces new syntax, and another a new analysis, the combination of the two extensions would be missing the implementation of this analysis for this syntax. Either the composition is then broken, glue code must be written to bridge this conflict, or there must be some mechanism to accurately and automatically generate this glue code.

Attribute grammars are capable of solving the expression problem by manually providing "glue code" that provides for evaluating new attributes on new productions. However, the expression problem can also be automatically resolved *without glue code* for attribute grammars that include *forwarding* [19]. An extension production that *forwards* to a "semantically equivalent" tree in the host language can evaluate new attributes introduced in other extensions via that host language tree, where the attribute will have defining semantic equations.

Although forwarding removes the need for the "glue code" necessary to resolve the expression problem, there are other ways in which a composition of attribute grammars may cause conflicts. Attribute grammars have a "well-definedness" property that, roughly speaking, ensures each attribute can actually be evaluated. However, although H, $H \triangleleft E_1$ and $H \triangleleft E_2$ may be well-defined, there is no guarantee that $H \triangleleft (E_1 \uplus_\emptyset E_2)$ will also be well-defined. As part of this thesis, we have developed a modular well-definedness analysis [11] that provides this guarantee. This analysis checks each $H \triangleleft E$ individually, and ensures that the composition $H \triangleleft (E_1 \uplus_\emptyset E_2)$ will be well-defined.

2.1 Related work

Domain-specific languages are traditionally implemented as an "external" DSL, and therefore incapable of composition with each other. *Internal* (or *Embedded* DSLs) are those implemented as a "mere" library in a suitable host language [9]. Internal DSLs are interesting in part because they permit the kind of composition we are interested in. However, they come with many drawbacks. For one, not all languages are practical choices for internal DSLs, including many that are in popular use, because the range of possible syntax is seriously limited by the host language. Further, in their simplest form, internal DSLs cannot easily perform domain-specific analysis, or complex translation.

One way of making internal DSLs capable of domain-specific analysis is to take advantage of complex embeddings into the host language's type system. AspectAG [21] and Ur/Web [2] are internal DSLs that take this approach to enforcing certain properties. The drawback to these approaches is the error messages: they are reported as type errors in the host language's interpretation of the types. In the worst case, understanding these error messages requires not just a deep understanding of the property being checked, but also the particular implementation and embedding of that property into the host language's type system.

One way to improve the ability of internal DSLs to generate code is to take advantage of meta-programming facilities in the language, like LISP macros, or

C++ templates. Racket [17] offers sophisticated forms of macros to enable this kind of translation. However, the static analysis capabilities of these macros are quite limited, though they are able to generate surprisingly good error messages for a macro system. (Especially surprising for those used to C++ template error messages.)

There are several systems for specifying languages that enable language extension and unification, as described in the introduction. JastAdd [7, 4], Kiama [16], and UUAG [1] are such systems based upon attribute grammars. SugarJ [6] is a recent system built upon SDF [8] and Stratego [22]. Rascal [12] is a meta-programming language with numerous high-level constructs for analyzing and manipulating programs. Helvetia [14] is a dynamic language based upon Smalltalk with language extension capabilities. However each of these systems requires that the composition of multiple language extensions may need to be repaired with glue code, and they otherwise provide little guarantee the composition will work. As a result, they do not support composable language extension, in our sense.

MPS [23] is a meta-programming environment that leans heavily on an object-oriented view of abstract syntax, and consequently struggles with expression problem in its support for composition. Consequently, the host language limits the possible analyses over syntax that are possible. Many useful language extensions do not necessarily need new analysis over the host language, however, as macro systems for dynamic languages already demonstrate.

3 Proposal

One component of this thesis has already been mentioned: our modular well-definedness analysis for attribute grammars [11]. This work is fully described elsewhere, but we will summarize it here. We say that an attribute grammar is *effectively complete* if, during attribute evaluation, no attribute is ever demanded that lacks a defining semantic equation. This analysis operates on each $H \triangleleft E$ individually, and provides an assurances that the resulting $H \triangleleft (E_1 \uplus_\emptyset E_2)$ will also have this property, without the need to explicitly check this composed language. To do this, the analysis is necessarily conservative about what extensions pass. Roughly speaking, extensions must satisfy the following requirements:

- Extensions must not alter the *flow types* of host language synthesized attributes. That is, they cannot require new (extension) inherited attributes be supplied in order to evaluate existing (host language) synthesized attributes.
- New productions introduced in extensions must *forward*.
- The flow types for new attributes introduced by an extension must account for the potential need to evaluate forward equations before they can be evaluated.

This modular well-definedness analysis, together with Copper's modular determinism analysis, offers a potential path towards composable language extension. Silver [18, 10] is an attribute grammar-based language with support for Copper, for which we have implemented our modular well-definedness analysis.

As the remainder of this thesis, we propose to evaluate whether this tool is truly capable of composable language extension. This is not a given, because the range of potential language extensions has been restricted:

- Forwarding requires all extensions' dynamic semantics be expressible in terms of the host language. We do not anticipate this restriction being a burden, as the host languages we're interested in extending are Turing-complete with rich IO semantics.
- Copper's analysis places restrictions on the syntax that can be introduced by extensions, relative to their host language. Again, since the host languages we are interested in extending often have highly complex concrete syntax already, we expect these restrictions will be a light burden.
- Silver's analysis places restrictions on how information can flow around abstract syntax trees. Again, however, this is relative to the host language implementation, which we expect to offer support for rich kinds of information flow already.

In light of these potential restrictions on the kinds of extensions that can be specified in Silver, we wish to validate each of our goals:

- The analyses themselves accomplish the goal of supporting composable language extension.
- We will need to implement at least two new extensions to the syntax.
- We will need to implement at least one new extension to static analysis.
- That static analysis extension should demonstrate the ability to generate good, domain-specific error messages.
- One of the extensions should involve either complex translation, require domain-specific optimizations, or have at least stringent efficiency requirements, to demonstrate the approach has little to no runtime overhead.

We propose to build a host language specification for C in Silver. C is an ambitious choice, but choosing a rich, practical language of independent design is necessary to evaluate whether the analyses' restrictions are practical, as they depend on the host language. To this specification of C, we propose to build language extensions that will meet the above requirements. These should ideally be language extensions that already exist in the literature, so that the changes to their design or syntax that are necessary to satisfy the analyses can be evaluated.

From this we hope to learn:

- How to better design extensible host language implementations, to support the development of interesting extensions. Many of the limitations imposed by the analyses depend upon the host language *implementation* more so than on the host language itself.
- Ways in which Silver itself may need to be extended to help specify the host language and extensions. For example, proper aggregation of error messages in the extensions could be ensured with language features specific to error message aggregation.

– Whether the restrictions still permit interesting and practical language extensions.
– Informally, whether the resulting extended languages are useful. We intend for our colleagues to make use of these extended languages, providing some feedback in this area, though we do not intend to perform an empirical investigation.

3.1 Future work

Beyond the scope of this thesis, there lie many more problems that must be solved to bring language extension to practicality.

First, host languages must be developed in Silver before they can be extended, and extensions can only be composed for a common host language, so fragmentation must be kept to a minimum to avoid splitting apart the ecosystem. High enough quality implementations of host languages for production use remains future work.

Second, numerous less daunting engineering issues would also need resolving. No obstacles to composing language extensions at runtime exist for Silver and Copper, but the feature has yet to be fully implemented. Further, the build process for making use of an extended compiler in large software projects must be worked out.

Third, a variety of other tooling must also be composable. Languages extensions must result not only in composed compilers, but also composed debuggers and integrated development environments. Abandoning these tools is not an option for practical use. We do not intend to directly address this problem in this thesis, though concurrent work for such tools in Silver is ongoing.

Finally, although these analyses ensure conflicts do not arise from the parser or attribute evaluator, it is possible that conflicts could arise in some other fashion. Certainly we can imagine blatantly wrong code, like suppressing all error messages from subtrees. But the formulation of *composable proofs* of the compiler's correctness would complete our understanding the problem posed by composable language extension.

4 Conclusion

We believe that no existing DSL implementation tool satisfies all five goals listed in the introduction: support composable language extension, allow extension to both and static analysis, provide good domain-specific error messages, and allow complex translation requirements. These goals are motivated by the desire to ensure that the users of language extensions can be certain they can draw on whatever high-quality extensions they need, without fear of breaking their compiler.

We have developed an analysis that ensures Silver meets the goal of supporting composable language extension, and we have implemented this analysis. We intend to develop an extensible specification of a popular and practical language, C, and we intended to demonstrate that practical language extensions to it are possible that satisfy this analysis. We believe this will demonstrate that Silver satisfies all five goals listed in the introduction for an ideal DSL implementation technique.

References

1. Baars, A., Swierstra, D., Loh, A.: Utrecht University AG system manual, http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem.
2. Chlipala, A.: Ur: statically-typed metaprogramming with type-level record computation. In: PLDI, 2010. pp. 122–133. ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806596.1806612
3. Deursen, A.v., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices 35(6), 26–36 (Jun 2000)
4. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: Proc. of ECOOP '04 Conf. pp. 144–169 (2004)
5. Erdweg, S., Giarrusso, P., Rendel, T.: Language composition untangled. In: LDTA, 2012 (2012)
6. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: OOPSLA 2011. pp. 391–406. ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2048066.2048099
7. Hedin, G., Magnusson, E.: JastAdd - an aspect oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
8. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf. SIGPLAN Not. 24(11), 43–75 (Nov 1989), http://doi.acm.org/10.1145/71605.71607
9. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4es) (1996)
10. Kaminski, T., Van Wyk, E.: Integrating attribute grammar and functional programming language features. In: Proceedings of 4th the International Conference on Software Language Engineering (SLE 2011). LNCS, vol. 6940, pp. 263–282. Springer (July 2011)
11. Kaminski, T., Van Wyk, E.: Modular well-definedness analysis for attribute grammars (2012), accepted SLE 2012
12. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proc. of Source Code Analysis and Manipulation (SCAM 2009) (2009)
13. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in **5**(1971) pp. 95–96
14. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: ECOOP 2010. pp. 380–404. Springer (2010)
15. Schwerdfeger, A., Van Wyk, E.: Verifiable composition of deterministic grammars. In: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press (June 2009)
16. Sloane, A.M.: Lightweight language processing in kiama. In: Proc. of the 3rd summer school on Generative and transformational techniques in software engineering III (GTTSE 09). pp. 408–425. Springer (2011)
17. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: PLDI 2011. pp. 132–141. ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1993498.1993514

18. Van Wyk, E., Bodin, D., Krishnan, L., Gao, J.: Silver: an extensible attribute grammar system. Scinece of Computer Programming (2009), accpeted, In Press
19. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proc. 11th Intl. Conf. on Compiler Construction. LNCS, vol. 2304, pp. 128–142 (2002)
20. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Intl. Conf. on Generative Programming and Component Engineering, (GPCE). ACM Press (October 2007)
21. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In: Proc. of 2009 International Conference on Functional Programming (ICFP'09) (2009)
22. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: Rewriting Techniques and Applications (RTA'01). LNCS, vol. 2051, pp. 357–361. Springer-Verlag (2001)
23. Voelter, M.: Language and ide modularization, extension and composition with mps. In: GTTSE 2011 (2011)