Reliable composition of domain-specific language features

Ted Kaminski
Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
tedinski@cs.umn.edu

Eric Van Wyk
Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
evw@cs.umn.edu

1. INTRODUCTION

What motivates us to reach for domain-specific languages as a solution is often the need for domain-specific *language features*, rather than a true need for an independent language in its own right. General-purpose languages offer only the most broadly applicable abstractions, and do not venture deep into application domains. The furthest they go are linguistic features for simple interfaces: `foreach`, `do/flatmap`, LINQ, `async/await`. Designers avoid more specialized features because choices are permanent, designs are hard, and different domains demand different trade-offs. They cannot hope to please everyone.

But a separate language ("external DSL") like YACC isn't the only option for introducing new domain-specific language features. Alternative approaches include:

- "Internal" DSLs (e.g. parser combinators), which approximate a DSL, but are implemented merely as a library within a sufficiently flexible host language.
- String embedding (frequently SQL, regex), where the DSL is inside a string within another language, perhaps with tooling that can recognize and analyze it.
- Macro systems, like Racket [8]. These can be extraordinarily flexible and powerful, but come with serious drawbacks we'll discuss shortly.
- A variety of other systems that have not yet achieved the same level of adoption in practice, notably including approaches like language extension (JastAdd [1]), techniques like projectional editing (mbeddr [12]), or type-based syntactic disambiguation (Wyvern [6]).

In this talk, we'll discuss our approach to introducing domain-specific language features via language extensions. Both language extensions and macros suffer from a similar problem: the inability to **reliably compose** together multiple independent extensions. We believe reliable composition is the key feature that can make language extension practical. Internal DSLs suffer many drawbacks, but to use two such DSLs in one program, one need only import both. This task is considerably more complicated for language extension, and prone to failures for macro systems. (One may be dismayed to discover that with Typed Racket [8] you are unable to "extend the language of types" with a macro.)

A practical system of language extension would:

- Allow **new syntax**. Just the composition of syntax from two extensions may cause conflicts.
- Allow **new analysis**. Combined with new syntax, we go beyond just the expression problem to the independent extensibility problem [13].
- Require **no glue code**. Bringing two extensions together should be as easy as it is for internal DSLs, and should not require the skills of a compiler engineer.
- Permit no conflict and **no interference** between two extensions. Everything should just work.

2. THE STATE OF OUR ART

We have a candidate solution achieving all of these aims, and more besides: we can seamlessly re-use host language syntax, and we can give extension language features the same user experience of native features, without the feeling of being tacked-on.

2.1 Syntax and parsing

Copper [11] is a context-aware LR(1) parser and scanner generator. By considering scanning and parsing holistically, rather than as separate phases, the scanner can use parsing context to only match tokens valid in that parsing context. This dramatically reduces the possibility of conflicts between extensions, as separate contexts can no longer cause any lexical conflict with each other.

Copper further implements a *modular determinism analysis* [7] that imposes modest restrictions on extensions, and as a result guarantees that the extensions will compose without serious conflict. In simplified form:

- A *bridge production* transitioning from host to extension must begin with a *marking token*, which clearly identifies the extension.
- Any re-use of host language nonterminals in extension syntax must not add new terminals to the follow set of the host language nonterminal.

As a result of these restrictions, the only possibility of conflict is overlapping marking terminals. But this turns out to be the same as for internal DSLs: you might have to rename or qualify imported names. So Copper supports the same mechanism.

2.2 Semantics and analysis

For analysis, we use a system based on attribute grammars [5] with *forwarding* [10]. Forwarding (like context-awareness) significantly reduces the problems associated with composing extensions. When one extension introduces a new attribute, and another extension introduces a new production, there is a problem at the intersection: how to compute a value for that attribute for that production? Forwarding allows productions to compute a semantically equivalent tree in the host language, similarly to macros, where that attribute can be evaluated instead.

Silver [9] is our attribute grammar-based programming language that comes with a *modular well-definedness analysis* [4] that likewise places modest restrictions on extensions, but guarantees that composing multiple extensions will result in a well-defined attribute grammar. These restrictions:

- Ensure that bridge productions all forward.
- Suitably freeze the *flow types* of all synthesized attributes (the set of inherited attribute they may use to compute their values).

While the story for Copper can stop here, for semantics we have further concerns about correctness. One extension may access an attribute and get a totally unexpected value from an equation from another extension, causing misbehavior.

And so we go further here and give an approach to achieving *non-interference* of language extensions through *coherence* [3]. Skipping most of the theory here, this again imposes restrictions on extensions, but ensures all will be well-behaved when composed. It turns out there's a very reasonable and effective way to enforce this via simple property testing. Each value computed for an attribute can be required to obey an equality such as:

```
t.typerep.host = t.host.typerep
```

This approach appears to be sufficient to catch non-malicious forms of non-compliance with the coherence restrictions.

2.3 Application

Using the above tools, we have implemented a C front-end compiler that supports reliably composable language extension, called AbleC [2]. AbleC supports C11 and many GCC extensions, and presently works by parsing, analyzing, and translating extended-language programs to plain C, before passing that off to an ordinary compiler (such as GCC).

```
cilk int count_matches(Tree *t) {
  match(t) {
    Fork(t1, t2, str) -> {
      int res_t1, res_t2, res_str;
      spawn res_t1 = count_matches(t1);
      spawn res_t2 = count_matches(t2);
      res_str = (str =~ /[1-9]+/) ? 1 : 0;
      sync;
      cilk return res_t1 + res_t2 + res_str;
    }
    Leaf(/[1-9]+/) -> { cilk return 1; }
    _ -> { cilk return 0; }
  };
}
```

Figure 1: Cilk, pattern matching, and regex independent extensions to AbleC used together.

3. THE ROAD AHEAD

Our approach has yielded what we believe to be a candidate, practical approach to introducing domain-specific language features via language extension. But there remains a lot of both research and development work left to do. For one thing, while we believe Silver & Coppper are a practical solution for implementing an extensible *compiler*, there are other important tools in language ecosystems.

For tooling like **debuggers** and **documentation generators**, we believe our approach can be made to adapt well with future development. The task here is simply one of ensuring that users are not confronted with generated code they did not write. This can be accomplished by designing the host language such that this information can be attached as desired, instead of computed from the (for extensions, machine-generated) AST.

A more vexing problem is IDE support. Some IDE features can be supported in a similar way as the above kinds of tools: the host language defines an API that extensions can use to get the correct behavior. It would be annoying that the host language needs polluting with such concerns, but acceptable. The trouble comes with features like **refactorings**. Refactorings likewise want to analyze the code as-written, not the code that gets generated. We'd like to be able to introduce new refactorings like we introduce new extensions, but it's not possible for the host language to support all possible refactorings in advance. It's not yet clear where to go from here; further research is required.

In this talk, we'll introduce an example of an extension to AbleC, go through how the restrictions on extensions influence its design, and discuss the maturity of our platform and future directions.

4. REFERENCES

[1] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

[2] T. Kaminski, L. Kramer, T. Carlson, and E. Van Wyk. Reliable and automatic composition of language extensions to C. *To appear OOPSLA 2017.*

[3] T. Kaminski and E. Van Wyk. Ensuring non-interference of composable language extensions. *To appear SLE 2017.*

[4] T. Kaminski and E. Van Wyk. Modular well-definedness analysis for attribute grammars. In *SLE 2012*, v7745 of *LNCS*, p352–371. Springer, 2012.

[5] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.

[6] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP 2014*, volume 8586 of *LNCS*, pages 105–130. Springer, 2014.

[7] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *PLDI 2009*, pages 199–210. ACM, 2009.

[8] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI 2011*, pages 132–141. ACM, 2011.

[9] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *SCP*, 75(1–2):39–54, January 2010.

[10] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC 2002*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.

[11] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE 2007*, pages 63–72. ACM, 2007.

[12] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. Mbeddr: An extensible C-based programming language and IDE for embedded systems. In *SPLASH Wavefront 2012*, pages 121–140, ACM, 2012.

[13] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *FOOL 2005.*