Scaling language specifications to mainstream languages and real-world applications ⋆

Ted Kaminski and Eric Van Wyk

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

Abstract. This paper describes two characteristics of language specification tools that support their use at scales beyond small prototypes. First is the ability to both explicitly and implicitly (via translation) specify the semantics of language constructs. In attribute grammars this achieved by *forwarding* and is used to specify languages by building features on top of a smaller core language. Second is the use of modular analyses on language specifications to guarantee that their eventual composition will have certain well-definedness properties.

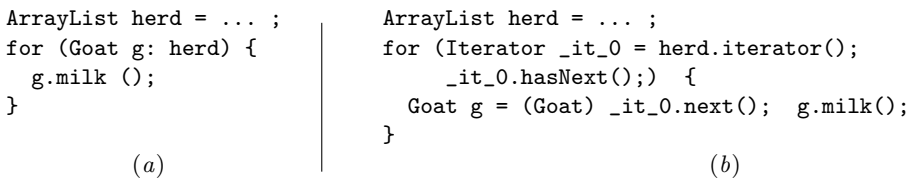
Forwarding: scaling up languages

One way in which language designers effectively build implementations for rich, full-featured programming languages such as Java, C, or C# is to build a manageable so-called *core* language and implement additional constructs such that they can be realized as (translated down to) combinations of constructs in the core language. Examples of this include translating a for-loop to an initializing assignment and while-loop, translating multiple declarations, such as `int x,y,z`, to a sequence of single declarations, such as `int x; int y; int z;`, and, as shown in Figure 1 (*a*) and (*b*), translating a Java 1.5 enhanced-for-loop to its equivalent Java 1.4 for-loop. This is a common practice as we do not want to implement all the semantics (name binding, type checking, optimization, and code generation) for these as that can become tedious.

To scale the use of declarative language specifications to provide high-quality implementations of such languages we have found that it helpful to be able to *explicitly* specify some of the semantics of the non-core constructs while specifying the remainder *implicitly* via translation to the core language. For example, we may want to perform some error-checking on the enhanced-for-loop construct explicitly but let its translation down to the traditional for-loop implicitly specify its translation to byte code.

This can achieved in attribute grammars [3] using a technique called *forwarding* [8] and is used regularly in Silver [7], our extensible attribute grammar system. What distinguishes forwarding from term rewriting and macros is that one can define some semantics explicitly, leaving the remainder to be defined

⋆ This work is partially supported by NSF Awards No. 0905581 and 1047961.



```
ArrayList herd = ... ;
for (Goat g: herd) {
  g.milk ();
}
```

(a)

```
ArrayList herd = ... ;
for (Iterator _it_0 = herd.iterator();
     _it_0.hasNext();)  {
  Goat g = (Goat) _it_0.next();  g.milk();
}
```

(b)

Fig. 1. Use of Java 1.5 enhanced-for statement (*a*), its translation to Java 1.4 (*b*).

implicitly by the translation down to a core language. Forwarding allows the production for the enhanced-for-loop to explicitly perform some static error checking by defining an *errors* attribute to check if the expression to be iterated over is a collection or an array. If it is not, an error message specific to the enhanced-for is generated. At attribute evaluation time, this production also constructs the appropriate syntax tree, something like in Figure 1 (*b*) (when the expression is a collection), and *forwards* any queries for attribute values that it does not explicitly define to that tree. Consider, for example, an attribute named *jvm* that decorates tree nodes and stores their translation to Java byte code. When there is no equation for this attribute on the production, the value of this attribute is automatically copied from that value of that attribute on the forwards-to tree. In this case, the translation to byte code for the enhanced-for-loop is simply the byte code computed for the regular for-loop that it forwards to.

As languages grow, both in the number of syntactic constructs and in the number of semantic analyses, optimizations, or translation targets, so does the need to *not* explicitly specify all semantics for all constructs. In ableJ, our extensible specification of Java, this loop and other Java 1.5 features, such as auto-boxing and unboxing, can be easily implemented using forwarding [9].

Modular analysis: scaling to real-world applications

We are interested in scaling up mainstream languages to include expressive domain-specific features, specifically in ways that allow programmers to pick and choose from a set of independently-developed language extensions that add new (perhaps domain-specific) language constructs, new semantic analyses, optimizations and/or translations to their host language. One example in ableJ [9] is the addition of SQL to Java as an extension so that SQL queries can be written directly in an extended Java program, parsed by the extended compiler, and statically checked for type errors. This sort of extension adds new syntax and new semantic analyses over that new syntax. Other extensions may add new analysis (*e.g.* advanced error checking) over existing language constructs.

Since declarative specifications (be they context free grammars, attribute grammars, or term rewrite rules) are naturally composable, this is possible. But the composition may not have properties that we want: a composed context

free grammar may be ambiguous or not in the LALR(1) class of grammars, the composed attribute grammar may be missing attribute definition and thus not be well-defined. For extensible languages to be viable in real-world applications this composition must "just work." If the composition occasionally fails to generate a working compiler or translator, then, we believe, extensible languages will continue to be treated as interesting, but not useful, academic pursuits.

What is needed are *modular* analyses that can be run by the language extension designer, the person who can understand error messages about parse tables or attribute equations. One such analysis is the *modular determinism analysis*. [4]. This analysis, performed by individually by the language extension writers; it checks the host language context free grammar (CFG^H) with an individual extension context free grammar (CFG_i^E). If each extension passes this modular analysis then the grammar composed of the host and all the extensions will be in the LALR(1) class of grammars and thus have no parse table conflicts. This analysis can be stated formally as follows:

$$(\forall i \in [1,n].\ \mathit{conflictFree}(CFG^H \cup CFG_i^E) \wedge\ \mathit{isComposable}(CFG^H, CFG_i^E)\ ) \\ \implies \mathit{conflictFree}(CFG^H \cup \{CFG_1^E, \ldots, CFG_n^E\})$$

This ensures that if the parse table of the composition $CFG^H \cup CFG_i^E$ is conflict free and that some additional restrictions (*isComposable*) hold, then the composition of the host and *all* such extensions is conflict free. These additional restrictions are on the structure of the grammar and derivative objects such as follow-sets [4]. Based on our experience these restrictions are quite reasonable and allow syntactically expressive language extensions to be specified. This analysis also ensures that there are no lexical ambiguities in the composed scanner. Previous work by Schwerdfeger and Van Wyk [4] describes this analysis in full and the appropriate related work not discussed here.

Another modular analysis is the *modular well-definedness analysis* [2] that ensures that the attribute grammar that specifies the semantics of a language composed from the host language and various extension is *well-defined*, also known as *complete*. This means that there are no missing attribute equations. Even in the presence of forwarding this is needed since productions are not required to "forward" and we must check that inherited attributes are defined where they are needed. The modular well-definedness analysis, called *modComplete*, provides the following guarantee:

$$(\forall i \in [1,n].\mathit{modComplete}(AG^H, AG_i^E)) \implies \mathit{complete}(AG^H \cup \{AG_1^E, ..., AG_n^E\}).$$

Like the modular determinism analysis, it is performed by an extension writer. If the analysis fails, the extensions writer can modify the extension attribute grammar to fix the detected problems. This analysis also performs several structural checks on the grammar and computes the flow-types of the attributes, determining what inherited attributes are needed to compute each synthesized attribute and imposes some restrictions on these to ensure that the composition of grammars will be well-defined. Our previous paper [2] on this analysis provides the full details, discusses the restrictions, and much of the related work in extensible languages.

Concluding remarks

Forwarding was originally developed as part of the Microsoft Research Intentional Programming (IP) project [5], but the semantics in that setting was not the same as in attribute grammars. In IP, syntax trees, such as for the body of the enhanced-for-loop, that were also used in the forwards-to tree were *not* given a new set of inherited attributes in their new context in the forward-to tree. While these inherited attributes often have the same values for both trees that is not always the case. We believe that this was an error in IP.

While forwarding is useful in specifying full-featured languages it is not required. Others have implemented such languages using declarative specifications without forwarding: the JastAdd Java compiler [1] implements Java 1.5 on top of a Java 1.4 implementation and uses attribute grammars without forwarding and the ASF+SDF system implemented variety of COBOL language processors [6].

While both modular analyses put some restrictions on the type of language extensions we have found them to still allow quite expressive extensions. To the best of our knowledge, our tools Silver [7] and Copper [10] are the only ones that provide such modular analyses. Modular analyses of some sort are needed to provide strong guarantees to programmers that the extensions they choose will in fact compose to create a working compiler. Only then can extensible languages enter the mainstream as a reliable tool for programming.

References

1. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proc. of OOPSLA*, pages 1–18. ACM, 2007.
2. T. Kaminski and E. Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proc. of Software Language Engineering (SLE 2012)*, volume 7745 of *LNCS*, pages 352–371. Springer-Verlag, September 2012.
3. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.
4. A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *Proc. of PLDI*, pages 199–210. ACM, June 2009.
5. C. Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.
6. M. van den Brand, M. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purpose. In *Proc. of 2nd Workshop on the Theory and Practice of Algebraic Specications*. Springer, 1997.
7. E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.
8. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. of Compiler Construction (CC 2002)*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.
9. E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *Proc. of ECOOP*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, 2007.
10. E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proc. of Generative Programming and Component Engineering, (GPCE 2007)*, pages 63–72. ACM Press, October 2007.