

Modular Domain-Specific Language Extensions*

Eric Van Wyk
University of Minnesota

1 Extensible languages and domain-specific extensions

Software development is a time-consuming, and error-prone process. A significant part of the problem is caused by the large semantic gap between the programmer’s high-level understanding of the problem and the relatively low-level language in which the problem solutions are encoded. General purpose language features such as generics in object-oriented programming or higher-order functions are helpful in specifying abstractions for a specific problem or domain, but these are limited in that they only provide the functionality of the desired abstractions. Domain-specific languages [6] can provide this same functionality but they also provide language constructs (notations) for the (domain-specific) abstractions and thus help reduce this semantic gap by raising the level of abstraction of the language to that of a specific domain. These languages also provide domain-specific analyses and optimizations that are difficult, if not impossible, to specify for programs written in general purpose languages. But the fundamental issue remains since problems often cross multiple domains and no language contains all of the general-purpose and domain-specific features needed to address every aspect of the problem. Thus, programmers cannot “say what they mean” but must encode their ideas as programming idioms at a lower level of abstraction.

It is our position that extensible languages provide a promising way to improve the software development process and software quality by reducing the semantic gap. An extensible language can easily be extended with the unique combination of domain-specific language features that raise the level of abstraction to that of a particular problem, even when the problem crosses multiple domains. A framework for language extensibility should have two significant facets. First, it should satisfy the requirement of *completeness*, that is, it should allow any language extension to fit seamlessly into the host language and be as well-developed as the host language constructs. In particular, the feature designer should be able to specify new language constructs together with their domain-specific semantic analyses and techniques for their optimization. Second, the extensibility framework should be *modular* in that it should allow a programmer to extend a language by choosing from a collection of previously defined features knowing only the functionality they provide. To understand the extensibility we seek, a distinction is made between the activities of implementing a language extension, performed by a domain-expert feature designer, and using language extensions to create an extended language, performed by a programmer.

2 Attribute grammars and forwarding

A promising approach toward realizing extensible languages as described above is based on attribute grammars extended with a mechanism called *forwarding* [8]. We have developed an attribute grammar specification language called Silver and built its supporting tools for attribute evaluation in order to study extensible languages and modular language extensions. In this approach, a *host language* is defined by an attribute grammar specification and language extensions are defined as attribute grammar fragments. These fragments

*This work was partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

contain new productions that specify new language constructs and new attributes and attribute definitions (on new and host language productions) that specify the analysis and optimization of these constructs. The activity of creating an extended language specification entails taking the union of the productions and attribute definitions in the host language and language extension specifications. This is done automatically by the framework tools so no “glue” code is needed to add the new features to the host language specification. This maintains the distinction between the feature designer who implements a language construct and the programmer who builds an extended language from a host language and language extension specifications.

Forwarding enables the automatic composition of host language and language extension specifications. Feature designers define semantic properties specific to the extension constructs explicitly via attribute definitions. Forwarding implicitly defines semantic properties by translating the new construct into the host language. It is a unifying technique that mimics common language extension techniques like macro expansion and simple term rewriting inside an attribute grammar framework. To use forwarding, a language construct production specifies a semantically equivalent construct that defines the semantics not explicitly defined by the forwarding construct. In attribute grammar terms, a production defines a distinguished attributed AST that provides default values for synthesized attributes not explicitly defined by the production. An example will clarify. Consider adding a simple *foreach* construct to Java¹ to iterate over Java Collections:

```
production foreach for::Stmt ::= elem::Id componentType::Type collection::Expr body::Stmt
  for.pp = “foreach ” + elem.lexeme + “ in ” + collection.pp + “ do ” body.pp ;
  for.errors = if collection.type.implements (Collection) then no-error else ... ;
  forwards to parse “{ ‘componentType.pp’ ‘elem’ ;
                    for ( Iterator ‘iter’ = ‘collection’.iterator(); ‘iter’.hasNext(); )
                      { ‘elem’ = ( ‘componentType.pp’ ) ‘iter’.next() ;    ‘body’ }”
  where iter = generate_new_unique_Id ()
```

This puts syntactic sugar on a popular programming idiom but also defines its own error checking semantics. The *foreach* production has a left-hand side *Stmt* non-terminal named *for* and right-hand side non-terminals *Id*, *Type*, *Expr*, and *Stmt* named as indicated. It explicitly defines the synthesized pretty-print *pp* and *errors* attributes allowing us to generate error messages using the code written by the programmer. It also specifies its *forwards-to* tree as the expected host language block/for-loop construct built by parsing the string and using the “unquote” operator (‘_’) to reference right-hand side subtrees, their attributes and the identifier *iter*. A left-hand side node generated by the production *foreach* is called the “forwarding-node.” When queried for an attribute that it does not explicitly define it passes that request to the “forwards-to” node, the block/for-loop construct, that returns the attribute’s value. Thus, we re-use all attributes defined on the block/for-loop except those with explicit overriding definitions. The reuse of existing language constructs means that feature designers do not have to know all the details (attributes) of the forwards-to constructs.

Like macro expansion and the rewriting process employed in MetaBorg [3], forwarding reuses the semantics of existing language constructs, but unlike these, forwarding productions also explicitly define semantics, as attributes, that here generate extension-level error messages, something most macro systems and MetaBorg do not do. Some modern macro systems, such as Maya [1] and JTS [2], do however provide specific error checking facilities. The extensions proposed here are what Standish [7] would call “paraphrase” extensions as the new constructs we introduce are expressed, via forwarding, in the host language. However, they are more than paraphrases as they explicitly define some semantics via attribute definitions.

3 Experience with language extensions

Here we briefly describe some extensible languages and language extensions that we have built in this framework. We have implemented subsets of C and Java as extensible languages in this framework and specified a number of domain-specific features as language extensions.

¹This feature was added to Java in version 1.5. This exemplifies the rigidity in monolithic traditional languages in that a simple feature like this cannot be easily added. Instead, it is not available to users until the language designer provides it.

Computational geometry extensions for C: Computational Geometry is a domain in which writing efficient robust programs is challenging. One factor is the possible overflow and underflow of fixed precision numeric representations supported by general purpose languages. Unbounded precision numbers can be implemented using libraries but these are relatively inefficient as the compiler cannot perform optimizations based on available domain-specific information. Many geometric algorithms, such as computing the convex hull of a set of 2D points, perform qualitative tests on geometric elements such as points and lines. These tests, called *primitives*, include checking, for example, whether a point lies to the left or right of a directed line in 2D space and are implemented by determining the sign of some expression. It is the overflow of intermediate results in computing the value of this expression that is the main problem with hardware supported numeric types. Thus, exact-precision types are needed only for intermediate results in expressions implementing primitives and not in variables whose value may be assigned inside of branch or loop statements. Therefore, static analyses such as determining the number of bits that will be required to store the exact-precision value can be performed.

We have built an extension to C that performs these analyses (and others also found in the domain-specific language LN [5]) and uses the results to generate efficient pure-C programs. The unbounded-precision intermediate values are stored in arrays. Because the static size of the intermediate values are known the upper bounds on loops that compute sums and products of these values are constants. Thus these loops can be unrolled to generate very efficient code. Primitives implemented in our language extension execute several times more quickly than primitives implemented by the the popular computational geometry library CGAL [4] that uses C++ template programming techniques. The primary reasons for this are the additional static analyses and optimizations that are possible in our framework and the avoidance of overhead in allocating objects representing unbounded precision values.

Adding SQL to Java: In a different extension we have implemented SQL as a language extension to Java. This provides many advantages over library based approaches such as the widely used Java DataBase Connector (JDBC) library. To implement an SQL query on a database in the library based approach, one may write the following:

```
int limit = 65;
rs = conn.createStatement().execute ("select Name from CustTable where Age > " + limit) ;
```

The syntax is clumsy and syntax errors, such as mis-spelling SQL keywords, and type errors, such as comparing a numeric field to a string, are not found until runtime. When SQL and some supporting language constructs are added to Java as a language extension, one can instead write the following:

```
int limit = 65; rs = on conn execute { select Name from CustTable where Age > limit } ;
```

This extension is implemented by defining productions (both concrete and abstract) for the SQL language and the “on ... execute” construct that takes a database connection and SQL query to perform. We specify attributes on the new SQL constructs that implement type checking in a standard way. This would forward to pure Java code very much like that shown in the library approach. The language extension approach has several advantages including the less clumsy syntax and compile time checking for syntax and type errors.

Combining Extensions: A programmer developing an application that performs geometric computations over data elements stored in a relational database would be able to import the SQL extension and (a Java version of) the computational geometry extension into an extensible implementation of the Java host language. This would allow the use, in a single program, of constructs from multiple domains. Both of these extensions perform semantic analyses on the new language constructs so that the optimizations and error-checking that are done in the domain-specific languages LN and SQL are performed by the language extensions in the compiler for the extended language.

4 Open issues

We raise two issues, the first focuses on extensible languages while the second has broader focus and is applicable to domain-specific languages in general.

If we are to allow programmers who are not language designers to import language features into an extensible host language then we must provide some guarantees that the features behave appropriately. Thus, what, precisely, does it mean to require that language extensions do not (adversely) interfere with one another? The extensions defined above are well-behaved in that the code they forward to and errors they generate in a particular program are not affected by the presence of constructs from the other extensions in the same program. But some semantics, such as the pretty print attribute, are affected by the presence of other extension; this seems appropriate however. Although attribute grammar fragments may be more amenable to “interference” analyses, how can we precisely specify what “adverse interference” or “safe composition of features” is? How can we define algorithms to automatically detect it for the programmer. We have some initial answers to these questions but are looking for more complete solutions.

A second issue has broader focus and asks how can we expect domain experts to effectively create domain-specific languages? Domain experts are often not programming language experts and may not be expected to effectively build and maintain compilers, analyzers, and debuggers for domain-specific languages. Similarly, as computer scientists, we may have expertise in programming language design but we often lack the domain expertise in computational geometry, database queries, or any number of other domains that must be understood to build domain-specific languages. Few people have expertise in both categories. We have faced this problem in building interesting domain-specific language extensions and are helped by domain experts with whom we can team up to build extensions. But not all domain experts have the opportunity to work directly with someone knowledgeable in language design and implementation. Providing a declarative high-level language like attribute grammar specifications languages to domain experts may be a step in the right direction but the fundamental challenge is only slightly mitigated by this.

References

- [1] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 270–281. ACM, 2002.
- [2] D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.
- [3] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. ACM Conf. on Object-oriented programming, systems, languages, and applications*, pages 365–383, 2004.
- [4] A. Fgabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The CGAL kernel: A basis for geometric computation. In *Proc. Workshop on Applied Computational Geometry*, May 1996.
- [5] Steven Fortune and Christopher J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, July 1996.
- [6] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
- [7] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21, 1975.
- [8] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.