

# Reusable Language Specification Modules in JastAdd II

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden  
(torbjorn|gorel)@cs.lth.se

**Abstract.** This paper discusses how to build domain specific languages (DSL) on top of a general purpose language using our language implementation tool JastAdd II. The specification formalism in JastAdd II is based on ReRAGs (Rewritable Reference Attributed Grammars) that combine object-oriented abstract grammars, static aspect-oriented programming, reference attributed grammars, and conditional rewriting. The technique is illustrated by evolving a matrix framework into new language constructs that are added on top of Java. The extension is done in a modular way and re-uses large part of the static semantic analysis from the base grammar.

## 1 Introduction

It is an interesting prospective to build domain-specific languages on top of general-purpose object-oriented languages. This can happen in several ways. One approach is to start with a general-purpose language and build a domain-specific framework. As the framework matures into a black-box framework, the application code will be mainly about configuring the blackbox framework, and a DSL for the framework configuration could be developed, as suggested by Roberts et al [RR96]. Such a DSL could be quite simple, declarative, and just used for generating the appropriate configuration code for the framework.

Another approach is also framework-based. Here, the domain-specific framework is just one part of a larger application, and it may be beneficial to evolve the domain-specific parts into new language constructs that are added to the general-purpose language. The benefits may be shorter and clearer code, static checking of programming conventions, and special-purpose optimizations. An example could be the development of a matrix-handling library, and the introduction of new language constructs for the library.

A third approach to building DSLs on top of GPLs is where the DSL incorporates GPL constructs as parts. An example could be the design of a new state-machine language, incorporating Java as the action language for state transitions.

In the two latter approaches, the DSL will be a large language, incorporating large parts, if not all, of a general purpose language. It would be highly beneficial to be able to reuse and easily extend the implementation of the GPL in order to implement the DSL. In this short paper we will briefly outline the mechanisms in our tool JastAdd II, and discuss how such reuse can be supported.

## 2 JastAdd II

JastAdd II is a language implementation tool that supports the generation of compilers and similar tools from grammars. It is based on ReRAGs (Rewritable Reference Attributed Grammars) that combine object-oriented abstract grammars, static aspect-oriented programming, reference attributed grammars, and conditional rewriting [EH04].

The input consists of an abstract grammar (`lang.ast`), and a number of ReRAG modules (`*.rag`). The abstract grammar defines a basic class hierarchy for the abstract syntax tree (AST). For example, general classes like `Decl`, `Stmt` and `Exp`, and more specific classes like `Assignment` and `ForStmt`. The ReRAG modules are aspects that add features to the AST classes in a way analogous to the static introduction feature of AspectJ [KHH<sup>+</sup>01]. In addition to fields, methods, and interface addition, ReRAGs support the introduction of attribute declarations, equations, and rewrite rules. The JastAdd implementation translates these to fields and methods that are woven into the AST classes. Typically, various computations on the AST are specified using ReRAG modules, e.g., name analysis, type analysis, error-checking, code generation, etc. The advantage of using ReRAGs to specify these computations (as compared to implementing them using imperative Java code) is that the formalism is declarative, and allows short and clear specifications. The executable implementation is automatically generated and is reasonably efficient (we have implemented a Java 1.4 [GJSB00] compiler that is around a factor of four in compilation time compared to the hand coded javac compiler).

The result of a ReRAG specification is an AST class hierarchy with an API for the specified computations. For example, the name analysis results in a method for identifier nodes that returns the corresponding declaration node. Similarly, the type analysis results in a method for `Exp` nodes that returns the type of the expression.

When building a tool with JastAdd, it is also possible to add ordinary Java code. This can be Java interfaces and classes used by the ReRAG modules, e.g., to implement the type system used by the type analysis module. Another use of ordinary Java code can be to simply use the generated ReRAG API. Such use can be specified as ordinary imperative Java aspects that further augment the AST classes. For example, an imperative code generator can be added that makes use of the name and type analysis API.

Our largest application of JastAdd II is currently a Java 1.4 front end that reads in Java source code, builds an annotated abstract syntax tree, performs static-semantic checking, and outputs pretty-printed source code again. This front end is a suitable base for experimenting with various language extensions, either general purpose or more domain-specific. The extensions can be described in a modular fashion, and there are mechanisms in JastAdd that support the translation of the new extended language into Java, reusing the existing Java implementation.

JastAdd II is not tied to any specific underlying parsing technology. It operates directly on ASTs and assumes that a separate parser is used for generating ASTs from source code. In our applications we have used both CUP (an LALR-based parser generator) and JavaCC (a LL(k)-based parser generator) as the parser component.

### 3 Examples of language extensions

As an example of a language extension, consider the addition of language constructs for matrix manipulation. As a first step, we might implement a matrix framework with methods for addition, multiplication, etc. As the next step, we would like to extend Java to support matrices as a built-in type, and where arithmetic operators like addition and multiplication are overridden to support also matrices.

The language needs to be extended in several ways. At the syntactic level, the language needs to be extended with a notation for matrix types and matrix literals. At the static-semantic level, we need to extend the type analysis to handle the new matrix types. To support the dynamic semantics we can simply translate the new language constructs to the equivalent in the matrix framework. E.g., translate a matrix literal to a series of object constructions, and a matrix addition to a method call on the matrix objects.

#### 3.1 Syntactic extensions

To support matrices in Java the parsing grammar is extended with the keyword *Matrix* and productions for matrix declarations, literals, and instantiation. An example that declares and instantiates a two by two matrix, *m*, where the elements are *integers* is shown below. The matrix is also initialized using matrix literal.

```
Matrix[int] m = new Matrix[2][2];  
m = [1 2]  
    [3 4];
```

The AST is extended with node types for matrix reference declarations, matrix instance expressions, and matrix literals. These nodes are all extending the behavior of a corresponding generic abstract node type. That way a lot of generic behaviour, that may be specialized, is re-used. New syntax may introduce ambiguities in that the same context-free syntax are used for language constructs that carry different semantic meaning. If so is the case, we build generic AST-nodes that are later rewritten during run-time when contextual information needed to resolve the semantic meaning is available. This technique has been successfully used in resolving syntactically ambiguous names in Java.

#### 3.2 Name analysis

When using the ReRAGs formalism, name binding is done by defining a reference attribute from each use of a name to its corresponding declaration node. The same type of binding is used for all names, e.g. variable-, type-, and package-names. Since all the added AST nodes inherit the behavior from their corresponding abstract node types, the existing name binding functionality is re-used without further modification.

### 3.3 Extending the type system

The type of an expression is represented by a reference to the corresponding type declaration. All declarations are subclasses of a generic type declaration, e.g. there are subclasses for *reference types*, *primitive types*, and *array types*. The type system is therefore extended with a new type, *matrix type*, that subclasses the generic type declaration. All types, including primitive types, are explicitly defined through a corresponding type declaration in the AST.

Each type has methods to determine if that particular type is compatible with the type given as a parameter. That functionality is overridden in the matrix type so that matrices are only compatible with other matrices. The elements must be numeric primitive types and the standard widening of primitive types if used to verify that two matrices are compatible.

### 3.4 Code generation

ReRAGs support conditional rewrites that may be defined for any AST node type. When a condition is true for a specific node, the subtree where that node is the root may be rewritten in a type-safe manner. The conditions may read attributes in a partially attributed tree, and context sensitive information may thus be used to determine when and how the subtree should be rewritten. This can be used to rewrite the Matrix extensions to use the framework, e.g. rewrites are added to binary expressions like addition and subtraction and the expression is rewritten to use the corresponding operation in the framework. The condition checks that there are no type errors and that the type of both involved operands is the matrix type. The same technique can be used to generate code for matrix declarations, literals, and instantiations.

## 4 Conclusions

In this paper we have discussed how existing language implementation can be re-used when evolving frameworks into DSLs adding language constructs to an existing languages. We have outlined how Java can be extended with a matrix language in a modular fashion using the ReRAGs language specification formalism. The object-oriented basis of ReRAGs gives a natural representation of the program where functionality can easily be re-used and extended using inheritance, overriding, and rewriting.

## References

- [EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, 2004. Accepted for publication.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
- [KHH<sup>+</sup>01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
- [RR96] D. Roberts and R.Johnson. Evolve frameworks into domain-specific languages. In *Proceedings of 3rd International Conference on Pattern Languages*, 1996.