A case study of separation of concerns in compiler construction using JastAdd II

Torbjörn Ekman
Department of Computer Science, Lund University, Sweden
torbjorn.ekman@cs.lth.se

Abstract

This paper presents a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. A domain-specific specification language, Rewritable Reference Attributed Grammars (ReRAGs), is combined with Java to implement compilers in a high-level declarative and modular fashion. Three synergistic mechanisms for separations of concerns are described: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Each technique is presented using a series of simplified examples from static semantic analysis for the Java programming language.

1 Introduction

We present a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. Simplified examples from static semantic analysis for the Java programming language [GJSB00] are used to demonstrate the mechanisms for separation of concerns provided in JastAdd II. This work is part of a larger project where the entire static semantics of Java 1.4 have been implemented. We believe that Java is a suitable language implementation for experimenting on language modularisation because of its advanced scope rules with nested types and inheritance as well as need for reclassification of contextually ambiguous names during name analysis.

JastAdd II uses a declarative compiler specification in the form of Rewritable Reference Attributed Grammars (ReRAGs) [EH04] combined with imperative Java code. ReRAGs provide three synergistic mechanisms for separations of concerns: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. This allows compilers to be written in a high-level declarative and modular fashion.

The rest of this paper is structured as follows. Section 2 describes JastAdd II and its specification language. The three mechanisms for separation of concerns are demonstrated in sections 3, 4, and 5. Section 6 discusses how the three mechanisms are used to deal with interaction between aspects. Section 7 points out some related work and Section 8 concludes this paper and discusses some future work.

2 JastAdd II Background

JastAdd II is an aspect-oriented compiler compiler tool using declarative Rewritable Reference Attributed Grammars (ReRAGs) and Java as its specification languages. The grammars define attributes and equations to specify computations and information propagation in the abstract syntax tree (AST). The formalism is object-oriented viewing the grammar as a class hierarchy and the AST nodes as instances of these classes. Behavior common to a group of language constructs can be specified in a common superclass and specialized or overridden for specific constructs in the corresponding subclasses. Often the most appropriate AST structure can only be decided after parial attribution of the AST. Rewrites allow restructuring of the tree to simplify the specification of the remaining attribution. The following sections give an introduction to JastAdd II compiler specifications.

2.1 The AST class hierarchy

The nodes in an Abstract Syntax Tree (AST) are viewed as instances of Java classes arranged in a subtype hierarchy similar to the Interpreter pattern, [GHJV95]. An

AST class correponds to a nonterminal or a production (or a combination thereof) and may define a number of descendents and their declared types, corresponding to a production right-hand side. In an actual AST, each node must be *type consistent* with its ancestor according to the normal type-checking rules of Java. I.e., the node must be an instance of a class that is the same or a subtype of the corresponding type declared in the ancestor. Shorthands for lists, optionals, and lexical items are also provided. All node types implicitly inherit the common ancestor type ASTNode that support generic access to node children. This is particular useful for generic tree traversals. An example definition of some AST classes is shown below.

```
// Expr corresponds to a nonterminal
ast Expr;

// Add corresponds to an Expr production
ast Add : Expr ::= Expr leftOp, Expr rightOp;

// Id corresponds to an Expr production
// id is a token
ast Id : Expr ::= <String id>;
```

2.2 Reference Attributed Grammars

ReRAGs are based on Reference Attributed Grammars (RAGs) which is an object-oriented extension to Attribute Grammars (AGs) [Knu68]. In plain AGs each node in the AST has a number of *attributes*, each defined by an *equation*. The right-hand side of the equation is an expression over other attribute values and defines the value of the left-hand side attribute.

Attributes can be *synthesized* or *inherited*. The equation for a synthesized attribute resides in the node itself, whereas for an inherited attribute, the equation resides in an ancestor node. Note that the term *inherited attribute* refers to an attribute defined in the ancestor node, and is thus a concept unrelated to the inheritance of OO languages. In this article we will use the term *inherited attribute* in its AG meaning, unless explicitly stated otherwise.

Inherited attributes are used for propagating information downwards in the tree, e.g. propagating information about declarations down to use sites, whereas synthesized attributes can be accessed from the ancestor and used for propagating information upwards in the tree, e.g. propagating type information up from an operand to its enclosing expression.

RAGs extend AGs by allowing attributes to have reference values, i.e., they may be object references to AST nodes. AGs, in contrast, only allow attributes to have primitive or structured algebraic values. This extension allows very simple and natural specifications, e.g., connecting a use of a variable directly to its declaration, or a class directly to its superclass. Plain AGs connect only through the AST hierarchy, which is very limiting.

In the JastAdd II implementation of RAGs attributes can be seen as methods where the method declaration and method body may be separated. Inherited attributes have their method body that defines the behavior in an ancestral node. An inherited attribute equation defines the behavior for a corresponding declaration of the same attribute in the subtree where the targeted equation node is the root. That way the only dependency on tree structure for that attribute is that the node holding the equation must be an ancestor to the node holding a declaration.

Aspects can be specified that define attributes, equations, and ordinary Java methods of the AST classes. An example is the following aspect for very simple type-checking.

```
// Declaration of an inherited attribute env
// of Expr nodes
inh Env Expr.env();

// Declaration of a synthesized attribute
// type of Expr nodes and its default equation
syn Type Expr.type() = TypeSystem.UNKNOWN;

// Overriding default equation for Add nodes
eq Add.type() = TypeSystem.INT;

// Overriding default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

The notation for method invocation is used when accessing descendent nodes like `leftOp` and `rightOp`, tokens like `id` and user-defined attributes like `env` and `type`. This API can be used freely in the right-hand sides of equations, as well as by ordinary Java code.

2.3 Rewrite rules

ReRAGs extends RAGs with rewrite rules that automatically and transparently rewrites nodes. The rewriting of a node is triggered by the first access to it. Such an access could occur either in an equation in the ancestor node, or in some imperative code traversing the AST. In either case, the access will be captured and a reference to the final rewritten tree will be the result of the access. This way, the rewriting process is transparent to any code accessing the AST.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps.

A rewrite rule for nodes of class *N* has the following general form:

```
rewrite N {
  when {cond}
  to R result;
}
```

This specifies that a node of type *N* may be replaced by another node of type *R* as specified in the result expression *result*. The rule is applicable if the (optional) boolean condition *cond* holds. Both the rewrite rule application order and the tree traversal order are implicitly defined by attribute dependences. A thorough description of ReRAGs implementation and application will appear in [EH04].

3 Inheritance for model modularisation

The subtype hierarchy generated from the grammar production rules provide excellent support for model modularisation. Generic behavior is defined in the possibly abstract node types and then specialized in the concrete node types. A small example adding a reference attribute to each expression referencing its corresponding type declaration node is shown below. The production rule hierarchy is in itself specialized in multiple steps, e.g binary operands, arithmetic expressions, and addive expressions are all succesive specializations from the generic language element expression. The type reference is defined to be boolean for all relational types while the type of arithmetic expressions is the widest type of both operands. The approach is generic in the sense that adding another arithmetic expression, e.g. subtraction, does not affect type propagation but merely requires implementation of the unique behavior, e.g. code generation.

```
ast Expr ;
ast BinOp : Expr ::= Expr left, Expr right ;

ast ArithmeticExpr : Binop ;
ast AddExpr : ArithmeticExpr ;

ast RelationalExpr : Binop ;
ast LessThanExpr : RelationalExpr ;

syn Decl Expr.type() ;
eq ArithmeticExpr.type() =
    widestType(left().type(), right().type());
eq RelationalExpr.type() = TypeSystem.BOOLEAN;
```

4 Aspects for cross-cutting concerns

The examples shown so far are actually feature aspects where attributes that cross-cut the AST subtype hierarchy are grouped into separate modules. This technique is very similar to static introduction techniques used in AspectJ [KHH+01], Hyper/J [OT01], and Multi Java [CLCM00].

The example below is a simple name binding module that binds a use-site to its declaration site through the inherited attribute `bind` taking a name as its parameter. A block of statements is modeled as a list of statements and a list of declarations for simplicity. Each block introduces a new scope to search for declarations and there are nested scopes since each statement in a block can be a block itself. The inherited attribute `bind` must thus have an equation in each scope, i.e. the `Block` node, and if a matching declaration is not found the search must be delegated to the surrounding scope.

```
ast Block : Stmt ::= Stmt stmt*, Decl decl*;
ast Name : Expr ::= <String name>;
ast Decl ::= <String name>;

protected inh Decl Name.bind(String name);
protected inh Decl Block.bind(String name);

eq Block.stmt().bind(String name) {
  for(int i = 0; i < numDecl(); i++)
    if(decl(i).name().equals(name))
      return decl(i);
  return bind(name);
}

public syn Decl Name.decl = bind(name());
```

To limit coupling between aspects such as name binding and type checking it is useful to limit visibility of certain attributes outside the defining aspect. The only attribute that needs to be exported outside a name binding aspect is for instance the binding from a use-place to its declaration, e.g. `decl` in `Name`. Attributes that define scope rules, e.g. `bind`, only affect the name binding and should thus be private to name binding modules.

Aspects have proven a very powerful technique to implement design pattern roles, [HK02], [NK01]. The same technique can be used in JastAdd II to implement reusable modules, illustrated below where the name binding approach described above is generalized in a generic module for nested scopes. The involved actors are nodes that need to lookup declarations and nodes that define new scopes. These actors are specified as interfaces and later used to tag each tree node that takes the role of an actor defined in the module. These interfaces also specify the equations that the implementors must supply to define non-generic behavior, e.g. finding declarations in its scope that matches the provided name. In the example, `Scope` represents nodes that defines a new scope and the non generic behavior is to match a name to a declaration while `Bind` represents the node that receives a reference to a declaration.

```
aspect NestedScopes {
  interface Scope {
    protected syn Decl lookup(String name);
  }

  interface Bind  {
    protected inh Decl bind(String name);
  }

  eq Scope.child().bind(String name) =
    lookup(name) != null ?
       lookup(name) : bind(name);
}
```

The module is generic in the sense that the only requirement on the AST structure is that an enclosing scope is defined by an ancestral node. It can be further generalized by adding more scope types, e.g. inheritance from super classes, and declare before use. Below is a name binding module that uses the module with the previously defined concrete node types `Block` and `Name`. The only behavior that needs to be implemented is the matching attribute `lookup` in `Block` and the use of the provided attribute `bind` in `Name`. In a Java several nodes implement a scope, e.g. block, class, interface, and for statement, and thus share common properties.

```
aspect NameBinding extends NestedScopes {
  declare parents: Block implements Scope;

  declare parents: Name implements Bind;

  eq Block.lookup(String name) {
    for(int i = 0; i < numDecl(); i++)
      if(decl(i).equals(name))
        return decl(i);
    return null;
  }

  public syn Decl Name.decl = bind(name());
}
```

5 Rewrites to create the most suitable model

Rewrites can improve separation of concerns by allowing computations to be expressed on the most suitable model. The information acquired during the early stages of static semantic analysis can be used to rewrite the model to make that information explicitly visible in the model structure for later stages.

We use an example from Java name analysis to demonstrate the technique. When parsing an expression containing qualified names, e.g. `java.lang.System.out`, it is syntactically undecidable if a part of a name is a reference to a package, type, field, or variable unless their context is taken into account. In the above example, `java` is most often a package, but only as long as there is no variable-, field-, or type-declaration named `java` that would shadow the package according to the Java scope rules. Thus, a context-free grammar can only build generic name nodes that capture all cases. The attribution will need to handle all these cases and therefore becomes complex. To avoid this complexity we would like to do *semantic specialization*, i.e. we would like to replace the general name nodes with more specialized ones. Other computations, like type checking, optimization, and code generation, can benefit from this rewrite by specifying different behavior in the specialized classes rather than having to deal with all the cases in the general name node.

An aspect that models Java names and resolves syntactically ambiguous names as described is shown below. There are two different types of names in Java from a syntactic point of view, simple names and qualified names. A simple name is a single identifier and a qualified name consists of a name, a `"."` token, and an identifier. During parsing a context-free grammar is used and thus general unbound names has to be build during AST creation. Semantic specialization is used to rewrite these general nodes into more specific ones, e.g. variable- or type-names. The ast-declarations in the aspect below model the described name structure.

Semantic specialization is implemented using a rewrite that rewrites an ambiguous `UnboundName` node into a `VariableName`-node or `TypeName`-node depend-

ing on the type of the binding received from the name binding module. Finally the `QualifiedName` nodes changes the scope rules for its right child to search the type of its left child to provide, e.g. when trying to bind out in the `System.out` expression the class `System` should be searched for a field named `out`.

```
ast Name : Expr;

ast SimpleName : Name ::= ID id;
ast QualifiedName : Name ::=
   Name left, SimpleName right;

ast UnboundName : SimpleName ;
ast VariableName : SimpleName ;
ast TypeName : SimpleName ;

// Resolve names depending on bound entity
rewrite UnboundName {
  when (bind().isVariableDecl())
  to SimpleName new VariableName(id());
  when (bind().isTypeDecl())
  to SimpleName new TypeName(id());
}

// The left name in a QualifiedName changes
// the scope for the name to the right
eq QualifiedName.right().bind(String name) {
  if(left() instanceof TypeName)
    return left().decl().lookup(name);
  if(left() instanceof VariableName)
    return left().type().lookup(name);
}
```

6 Aspect interaction

While the aspects demonstrated to far define static features we also use more pluggable aspects, e.g. a declare before use aspect to complement the name binding module in Section 4 and optional code optimization aspects. Pluggable aspects define rewrites that change a run-time node instance to a subtype node with extended behavior. That way an aspect can be added to the system in a way transparent to other aspects.

The examples demonstrated so far deal with equations that cross-cut the type hierarchy only and not cross-cutting concerns within equations. To override and extend attribute equations we use inheritance of the model structure in combination with rewrites that change the type of a node instance at run-time. I.e. we may have different/extended equations for UnboundName and VariableName defined in the example in Section 5. This technique, using run-time rewriting and inheritance, is more powerful than static compile-time point-cuts within equations in that it may take run-time information into account but less powerful in that each node may only be changed by a single aspect. Therefore it would be interesting to combine the current approach with more fine-grained static point-cuts within equations.

7 Related work

The introduction of attribute definitions and equations to an exisiting class hierarchy in a modular fashion used in JastAdd II is very similar to static introduction in AspectJ [KHH+01], hyperslices in Hyper/J [OT01], and open classes in MultiJava [CLCM00]. A functional approach to attribute grammar aspects using the same technique is presented in [dPJV00] where aspects are first-class objects that can be freely combined using a combinator library in Haskell. ReRAGs further improved modularisation support in that the current model instance may be rewritten during run-time to a more suitable model allowing each computation to be expressed on the most suitable model and more fine-grained separation of concerns within equations.

The Visitor pattern, [GHJV95], is often used in compiler construction for separation of concerns when using object-oriented languages. Visitors can only separate cross-cutting methods while static introductions can be used for fields as well. AOP implementations of the Visitor pattern need not rely on a delegation mechanism resulting in a cleaner more intuative implementation, [HK02]. ReRAGs aspects differs from AOP implementations of the Visitor pattern in that an explicit traversal strategy in the form of a Visitor is not specified but merely implicitly defined by attribute dependences. Rewrites further improves modularisation in that the underlying structure may change during run-time to better fit the current computation.

Higher order attribute grammars (HAGs) [VSK89] adds tree nodes computed reading the partially attributed AST at run-time and can thus provide a more suitable model. The process is, however, not transparent to other computations and is thus less flexible from a separation of concerns view. The use of attribute grammars and forwarding for modular language implementation is discussed in [VWMBK02]. Forwarding overrides attribute equation dynamically at run-time and forwards equation to a different part of the tree. Since it is based on HAGs the target tree can be computed at run-time and the approach is thus similar to semantic specialization.

8 Conclusions and future work

We have demonstrated three synergistic mechanisms for separations of concerns supported by ReRAGs in the JastAdd II compiler compiler: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Examples inspired by static semantic analysis of the Java programming languages have been used to illustrate and motivate each technique. We believe that this allows compilers to be written in a high-level declarative and modular fashion.

Our experiences indicate that the implementation leads to flexible solutions to several traditional compiler construction problems, and we hope to generalize some of these techniques and document them as design patterns or frameworks for compiler construction using ReRAGs.

We would also like to investigate the interaction between pluggable aspects and also how to better support fine-grained cross-cutting within equations combining AspectJ-like point-cuts with run-time rewriting implemented using ReRAGs in JastAdd II.

References

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of OOPSLA 2000*, volume 35(10), pages 130–145, 2000.

[dPJV00] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, 2004. Accepted for publication.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of OOPSLA-02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, November 4–8 2002. ACM Press.

[KHH+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.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[NK01] N. Noda and T. Kishi. Implementing design patterns using advanced separation of concerns. In *OOPSLA2001 workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[OT01] Harold Ossher and Petri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 23rd international conference on Software engineering*, pages 821–822. IEEE Computer Society, 2001.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.

[VWMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8-12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.