Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 253 (2010) 205–219
www.elsevier.com/locate/entcs

A Pure Object-Oriented Embedding of Attribute Grammars

Anthony M. Sloane[1,2]

Department of Computing, Macquarie University, Sydney, Australia

Lennart C. L. Kats, Eelco Visser[1,3]

Software Engineering Research Group, Delft University of Technology, Delft, The Netherlands

Abstract

Attribute grammars are a powerful specification paradigm for many language processing tasks, particularly semantic analysis of programming languages. Recent attribute grammar systems use dynamic scheduling algorithms to evaluate attributes by need. In this paper, we show how to remove the need for a generator, by embedding a dynamic approach in a modern, object-oriented programming language to implement a small, lightweight attribute grammar library. The Kiama attribution library has similar features to current generators, including cached, uncached, circular, higher-order and parameterised attributes, and implements new techniques for dynamic extension and variation of attribute equations. We use the Scala programming language because of its combination of object-oriented and functional features, support for domain-specific notations and emphasis on scalability. Unlike generators with specialised notation, Kiama attribute grammars use standard Scala notations such as pattern-matching functions for equations and mixins for composition. A performance analysis shows that our approach is practical for realistic language processing.

Keywords: language processing, compilers, domain-specific languages

1 Introduction

The language processing domain concerns the construction of compilers, interpreters, code generators, domain-specific language implementations, refactoring tools, static code analysers and other similar artefacts. Attribute grammars are a powerful processing formalism for many tasks within this domain, particularly for semantic analysis of programming languages [7,22].

[1] This research was supported by NWO projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, *TFA: Transformations for Abstractions*, and 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*.

[2] Email: `Anthony.Sloane@mq.edu.au`

[3] Email: `L.C.L.Kats@tudelft.nl`, `visser@acm.org`

1571-0661 © 2010 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2010.08.043

Attribute grammars extend context-free grammars with declarative equations that relate the values of attributes of grammar symbols to each other. Most attribute grammar systems translate the equations into an implementation written in a general purpose programming language. The translation makes decisions about attribute evaluation order and storage, removing the need for manual techniques such as visitors. Therefore, the developer is freed to focus on the language properties that are represented by the attributes.

In recent years, attribute grammar systems have focused on dynamically scheduled evaluation, where the attributes to be evaluated and the evaluation order are determined at run-time rather than at generation time [14]. LRC [23], JastAdd [11], UU AG [2], and Silver [25] are prominent examples of this approach. A dynamic schedule has the advantage that attributes are evaluated at most once, but adds runtime overhead. In applications such as integrated development environments, the tradeoff is particularly worthwhile, since not all attributes are needed at all times.

Nevertheless, these recent systems are based on generators that add to the learning curve and complicate the development and build processes. We show in this paper how to integrate a dynamically scheduled attribute grammar approach as a library into an existing modern, object-oriented language. We use a *pure embedding* where the syntax, concepts, expressiveness and libraries of the base language are used directly [12,18]. The high-level declarative nature of the attribute grammar formalism is retained and augmented with the flexibility and familiarity of the base language, both for specification and for implementation of the formalism itself.

This work is part of the Kiama project [24] [4] that is investigating pure embedding of language processing formalisms into the Scala programming language [20]. The main reasons for using Scala are its inclusion of both object-oriented programming and functional programming features, support for domain-specific notations, emphasis on scalability and interoperability with the Java virtual machine.

Kiama's attribution library has the same general power as systems such as JastAdd [11]. [5] Abstract syntax trees are defined by standard Scala classes with only minimal augmentation of the class definitions required to prepare them for attribution. Attribute equations are written as pattern matching functions of abstract tree nodes. As well as basic synthesised and inherited non-circular attributes, Kiama currently supports reference attributes [10], higher-order or non-terminal attributes [26], parameterised attributes [8], and circular attributes that are evaluated to a fixed point [16]. Language extension and modification are achieved using Scala's scalability constructs such as traits and mixins. Also, in contrast to previous systems, attribute definitions can be adapted at run-time to implement dynamic language variations. Overall, the performance of Kiama attribute evaluators is similar to dynamically scheduled evaluators produced by generators.

The rest of the paper is structured as follows. Section 2 provides an introduction to the features of Kiama's attribution library by way of two typical examples. The

[4] http://plrg.science.mq.edu.au/projects/show/kiama

[5] Like JastAdd, Kiama also has facilities for tree rewriting, but they are beyond the scope of this paper.

Kiama implementation is outlined in Section 3. Section 4 considers how language extension and separation of concerns can be achieved by leveraging the general Scala platform. We evaluate the performance of Kiama in Section 5. The paper concludes with a discussion of our approach in the context of other attribute grammar systems in Section 6 and concluding remarks in Section 7.

2 Attribute Grammars in Kiama

This section presents a couple of well-known examples to introduce the basic capabilities and style of the Kiama attribution library.

2.1 Repmin

Repmin is a classic problem of tree analysis and transformation, originally employed to illustrate the use of lazy circular programs in functional programming to eliminate multiple tree traversals [3]. Repmin is often used as a simple test of attribute grammar systems. The problem is to take a binary tree with integer leaves and transform it into a tree with the same structure, but with each leaf value replaced by the minimum leaf value of the original tree.

Kiama is intended to work as seamlessly as possible with a developer's non-Kiama Scala code, including libraries. Attribution is performed on trees made from standard Scala *case class* instances. A case class allows instances to be created without the usual `new` operator, provides structural equality and supports structure-based pattern matching. In this sense, case classes provide capabilities that are similar to algebraic data types found in languages such as Haskell and ML.

Figure 1(a) shows the abstract syntax for Repmin in Scala and a typical problem instance. Each class inherits from the `Attributable` Kiama library class to obtain generic functionality, but otherwise no changes are necessary. The case classes can have other fields, members, supertypes and so on, without affecting the attribution.

Figure 1(b) shows the definitions of the `locmin` (local minimum), and `globmin` (global minimum) integer-valued attributes and the `repmin` tree-valued attribute. (In this example, no attributes of `repmin` are demanded, but they could be, making it a higher-order attribute.) `attr` is a Kiama library function that takes as argument the attribute equations defined by cases on the node type. Each resulting attribute is a function from a node type to the type of the attribute value. Attributes in modular specifications should be partial functions to allow for composition, so Kiama constructs the type of an attribute using its own `==>` partial function type constructor instead of the usual Scala (total) function type constructor `=>`.

The pattern matching abilities of case classes are used in the attribute equations. Identifiers beginning with a lowercase letter are binding occurrences, whereas those beginning with an uppercase letter are constants. An underscore pattern matches anything. A `v @ p` pattern binds the name `v` to the value matched by the pattern `p`. A guard `if boolexp` matches if the expression `boolexp` evaluates to true.

On the right-hand side of an equation, attributes are accessed using a reference style: the value of attribute `a` of node `n` is written `n->a`. The definition of `globmin`

```
abstract class Tree extends Attributable
case class Pair (left : Tree, right : Tree) extends Tree
case class Leaf (value : Int) extends Tree

// repmin (Pair (Leaf (3), Pair (Leaf (1), Leaf (10))))
//       == Pair (Leaf (1), Pair (Leaf (1), Leaf (1)))
```

(a) Scala abstract syntax for Repmin trees and a simple problem instance.

```
val locmin : Tree ==> Int =
    attr {
        case Pair (l, r) => (l->locmin) min (r->locmin)
        case Leaf (v)    => v
    }

val globmin : Tree ==> Int =
    attr {
        case t if t isRoot => t->locmin
        case t             => t.parent[Tree]->globmin
    }

val repmin : Tree ==> Tree =
    attr {
        case Pair (l, r)  => Pair (l->repmin, r->repmin)
        case t @ Leaf (_) => Leaf (t->globmin)
    }
```

(b) Kiama attribute grammar for Repmin.

Fig. 1. A Kiama solution to the Repmin problem.

uses pre-defined *structural properties* to inspect the tree structure: `t isRoot` is true if `t` is the root of the tree and `t.parent` is a reference to `t`'s parent. (Scala allows the period in a method call `o.m` to be omitted, so `t isRoot` is just `t.isRoot`, and similarly for the call of the `min` method.) Note that since the parent has a generic type, it must be cast to a `Tree`. Section 6 revisits the typing question.

Overall, Repmin is defined in Kiama in a clear and natural way using mostly standard Scala features. Specialising an equation for a particular node type is easy using pattern matching. Defining more complex grouping of attribution is also straight-forward. For example, the definition of `globmin` applies at all nodes and propagates the root value down the tree in a modular fashion, without requiring voluminous copy rules or special constructs as in some other systems.

2.2 Variable liveness

Attribute grammars were originally designed to express computations on tree structures. With the addition of remote node references that follow naturally from an object-oriented representation of attribute grammars, graph-based algorithms can also be expressed [10]. For instance, references allow attributes to define a control flow graph. Furthermore, using fixed point iteration to evaluate attributes, attribute grammars can be used to express data-flow equations [16]. We illustrate these capabilities using a variable liveness computation for a simple imperative language [19].

Figure 2(a) shows a typical variable liveness problem instance, where the *In* and *Out* sets are the live variables reaching or leaving each statement. Figure 2(b)

```
                        In          Out
{
    y = v               {v, w}      {v, w, y}
    z = y               {v, w, y}   {v, w}
    x = v               {v, w}      {v, w, x}
    while (x) {         {v, w, x}   {v, w, x}
        x = w           {v, w}      {v, w}
        x = v           {v, w}      {v, w, x}
    }
    return x            {x}
}
```

(a) A variable liveness problem instance.

```
type Var = String
abstract class Stm extends Attributable
case class Assign (left : Var, right : Var) extends Stm
case class While (cond : Var, body : Stm) extends Stm
case class If (cond : Var, tru : Stm, fls : Stm) extends Stm
case class Block (stms : Stm*) extends Stm
case class Return (ret : Var) extends Stm
case class Empty () extends Stm
```

(b) Scala abstract syntax definition for liveness problem.

Fig. 2. The variable liveness problem.

shows the abstract syntax that is used for this example. In the definition of the `Block` class, the type `Stm*` is standard Scala that indicates that the `stms` field is a sequence of zero or more statements, implemented by the Scala collection library.

The liveness sets for a statement s are calculated from the variables defined by $s(\mathit{defines})$ and the variables used by $s(\mathit{uses})$ by iterative application of the standard data flow equations $\mathit{in}(s) = \mathit{uses}(s) \cup (\mathit{out}(s) \setminus \mathit{defines}(s))$ and $\mathit{out}(s) = \bigcup_{x \in \mathit{succ}(s)} \mathit{in}(x)$, where $\mathit{succ}(s)$ denotes the control-flow successors of s.

Figure 3 shows the Kiama definitions of these attributes. The control flow successor `succ` is a reference attribute defined in terms of a `following` attribute that defines the default linear control flow. `following` is defined as an inherited attribute by pattern matching on the parent node, using the convenience function `childAttr`. The `_*` patterns in these definitions match possibly-empty sequences.

The `circular` Kiama library function used in the definitions of *in* and *out* is like `attr` except that it also takes an initial value for the attribute and evaluates until a fixed point is reached. An alternative attribute access notation `a (n)` has been used for the liveness sets to emphasise the correspondence with the data flow equations. In the definition of *out*, the Scala library method `flatMap` applies *in* to each of the statement's successors and concatenates the results.

As in the Repmin example, the variable liveness definitions are relatively easy to follow, use mostly standard Scala, and correspond closely to the mathematical definitions of the various properties. The small Kiama attribution library interface is summarised in Figure 4. The next section outlines its implementation.

```
val succ : Stm ==> Set[Stm] =
    attr {
        case If (_, s1, s2)    => Set (s1, s2)
        case t @ While (_, s)  => t->following + s
        case Return (_)        => Set ()
        case Block (s, _*)     => Set (s)
        case s                 => s->following
    }

val following : Stm ==> Set[Stm] =
    childAttr {
        case s => {
            case t @ While (_, _)              => Set (t)
            case b @ Block (_*) if s isLast    => b->following
            case Block (_*)                    => Set (s.next)
            case _                             => Set ()
        }
    }

val uses : Stm ==> Set[String] =
    attr {
        case If (v, _, _)     => Set (v)
        case While (v, _)     => Set (v)
        case Assign (_, v)    => Set (v)
        case Return (v)       => Set (v)
        case _                => Set ()
    }

val defines : Stm ==> Set[String] =
    attr {
        case Assign (v, _) => Set (v)
        case _             => Set ()
    }

val in : Stm ==> Set[String] =
    circular (Set[String]()) {
        case s => uses (s) ++ (out (s) -- defines (s))
    }

val out : Stm ==> Set[String] =
    circular (Set[String]()) {
        case s => (s->succ) flatMap (in)
    }
```

Fig. 3. Kiama attribute grammar for the variable liveness problem.

3 Implementation

The Kiama implementation consists of two main parts: definitions of structural properties and evaluation mechanisms for the different kinds of attribute. The implementation of Kiama consists of about 230 lines of Scala code.

3.1 Structural Properties

Case classes to be attributed must inherit from the `Attributable` trait. Each case class is automatically an instance of Scala's `Product` trait that provides generic access to its constructor fields. The code that initialises an `Attributable` instance uses the `Product` interface to set the structural properties, such as `parent`, and, for nodes in sequences, `next` and `prev`.

A complication is that the attribution library must coexist with Scala code that processes the same data structures. In particular, nodes might contain sequences

`Attributable`	Supertype of all node types.
Structural attributes of all nodes	
`t.parent : Attributable`	Parent of `t`.
`t.isRoot : Boolean`	Is `t` the root of the tree?
Structural attributes of nodes occurring in sequences of nodes of type T	
`t.prev, t.next : T`	Siblings of `t`.
`t.isFirst, t.isLast : Boolean`	Is `t` the first or last node?
`t.index : Int`	Number of siblings before `t`.
For node type T, user-defined attributes of type U	
`attr (f : T => U) : T ==> U`	Basic attribute defined by `f`.
`circular (init : U) (f : T => U) : T ==> U`	Circular attribute defined by `f` with initial value `init`.
`childAttr (f : T => Attributable ==> U) : T ==> U`	Attribute defined by matching on parent.
`paramAttr (f : S => T ==> U) : S => T ==> U`	Attribute with parameter of type `S`.
Access attribute a of node n	
`n->a`	Reference style.
`a (n)`	Functional style.

Fig. 4. Summary of the Kiama attribution interface.

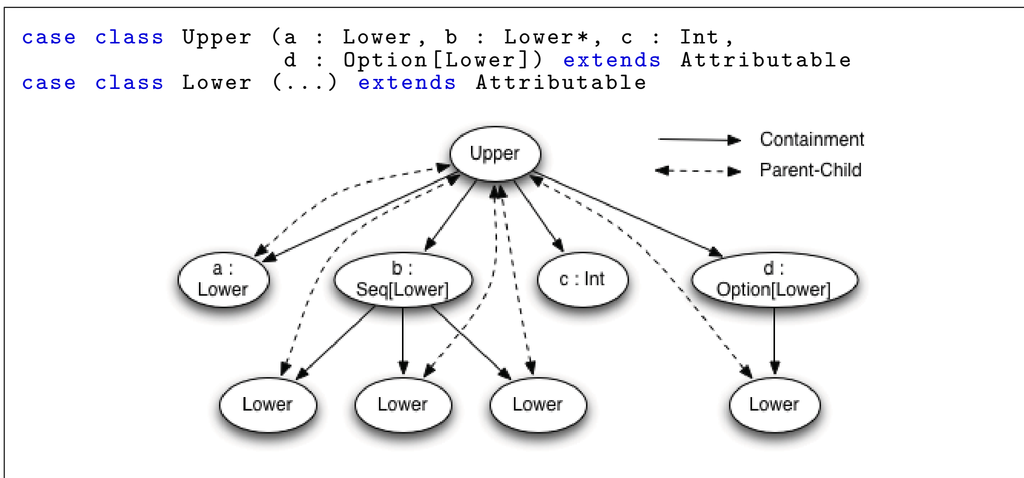

Fig. 5. The Kiama parent-child relation compared to structure containment.

and optional fields represented by Scala values of type `Seq[T]` and `Option[T]`. (`Option[T]` is analogous to Haskell's `Maybe a` type, having values of `None` or `Some (t)`, for some value `t` of type `T`.) Fields that are not attributable might also be present, most notably primitive values.

To address these issues, Kiama makes a distinction between the *containment relation* between a node and its fields as defined by the case class declaration, and the *parent-child relation* that relates an `Attributable` node to its `Attributable`

```
class CachedAttribute[T,U] (f : T ==> U) extends (T ==> U) {
    val memo = new IdentityHashMap[T,Option[U]]

    def apply (t : t) : U =
        memo.get (t) match {
            case None            => memo (t) = None
                                    val u = f (t)
                                    memo (t) = Some (u)
                                    u
            case Some (Some (u)) => u
            case Some (None)     => error ("Cycle detected")
        }

    def isDefinedAt (t : T) : Boolean = f isDefinedAt t
}
```

Fig. 6. The CachedAttribute class.

children. Both of these relations are useful in attribute equations. Figure 5 shows an example where an `Upper` node contains four fields: one required `Lower`, a sequence of zero or more `Lower` nodes, an integer and an optional `Lower`. The `Upper` node therefore has five `Attributable` children and those nodes have the `Upper` node as their parent. Most accesses to nodes in equations are performed via fields or the `parent` property, but Kiama also provides an iterator so that all `Attributable` children can be accessed in a generic way.

3.2 Attributes

Attributes defined by `attr` are implemented by the `CachedAttribute` class that we focus on here. Since attribute equations are cached and are not evaluated until they are needed, the evaluation method is equivalent to those used in early attribute grammar systems [14] and, more recently, in JastAdd [11].

Figure 6 shows the definition of the `CachedAttribute` class. The type parameters `T` and `U` denote the type of the nodes to which this attribute applies and the type of the attribute value, respectively. The value parameter `f` is the user-specified (partial) function that defines the attribute equations. Since `CachedAttribute` is a sub-class of the partial function type `T ==> U` and Scala converts `a (n)` into `a.apply (n)`, this implementation presents a convenient functional interface to the attribute value. The reference notation `n->a` is a simple alias.

The partial function implementing an attribute must define two methods: `apply`, that "runs" the defining equations on the given node and returns the value, and `isDefinedAt`, that provides information about the function's domain. For a cached attribute, `apply` uses a local hash map to memoise the attribute value for the node `t`. A marker value `None` is used to detect when the method calls itself, so that an error can be reported. The `isDefinedAt` method simply delegates to the `isDefinedAt` of the attribute equations.

Other kinds of attributes are defined by similar classes with the same interface. For example, uncached attributes are a simple variant. `circular` uses a `CircularAttribute` class that provides a functional interface to the fixed-point evaluation algorithms of Magnusson and Hedin [16].

It is sometimes useful to have attributes that are parameterised by other values. For example, JastAdd specifications often use this style in name analysis where a `lookup` attribute is parameterised by the name being sought [8]. Parameterised attributes are created in Kiama using the `paramAttr` function (Figure 4). For example, an attribute for looking up a name `n` can be defined in Kiama as follows.

```
val lookup : String => Attributable ==> Decl =
    paramAttr {
        n => {
            case ...  // cases for lookups at different nodes
        }
    }
```

4 Language Extensions and Separation of Concerns

Many attribute grammar systems allow for a high degree of separation of concerns, allowing different equations for an attribute or production to be defined across different modules. Typically, this modularity is implemented as a purely syntactic feature, joining together all equations for an attribute before compilation, and considering the entire, merged specification as a whole. (This is notably not the case for first-class attribute grammars [5], where attributes are *first-class citizens* and can be manipulated in the language.)

While other attribute grammar systems often use a general-purpose language for the expressions in attribute equations (e.g., Haskell in UU AG [2], Java in JastAdd), they provide their own module systems on top of that language. Kiama relies purely on Scala for the modular specification of attribute grammars. As a modern object-oriented programming language aimed at high-level abstraction for building modular frameworks with a rich, often functional interface, Scala offers an impressive toolbox of modularization features, most notably traits and mixins.

4.1 Static Separation of Concerns Using Traits

Flexible static combination of attribution modules can be achieved using Scala traits to define components and performing mixin composition to combine them [21]. For example, we can decompose the variable liveness problem of Section 2.2 into three components dealing with control flow, variables, and the liveness computation itself. The first two of these can be abstracted by interfaces defined by traits.

```
trait ControlFlow {
    val succ : Stm ==> Set[Stm]
}

trait Variables {
    val uses : Stm ==> Set[String]
    val defines : Stm ==> Set[String]
}
```

An implementation of the liveness module can use a Scala *self type* [21] to declare that it must be mixed in with implementations of the `ControlFlow` and `Variables` interfaces.

```
trait LivenessImpl extends Liveness {
    self : Liveness with Variables with ControlFlow =>
    ...
```

```
import kiama.attribution.DynamicAttribution._

case class Foreach (cond : Var, body : Stm) extends Stm

object DataflowForeach {
    Dataflow.succ += {
        case t @ Foreach (_, body) => following (t) + body }
    }

    Dataflow.following +=
        childAttr {
            _ => {
                case t @ Foreach(_, body) => following (t) + body
            }
        }
    }
}
```

Fig. 7. Dynamic attribute grammar extension.

```
        ... definitions of in and out as before
        ...
  }
```

Finally, an implementation of the dataflow solution can be formed by mixing together implementations of the three modules.

```
object Dataflow extends LivenessImpl with VariablesImpl
    with ControlFlowImpl
```

This approach allows modules to be composed with alternative implementations without being changed or even recompiled because the types ensure that the composition is valid.

4.2 Dynamically Extensible Attribute Definitions

Kiama uses functions to implement attributes, represented by `CachedAttribute` and other types. In this subsection, we illustrate the flexibility of this approach by adding a new dynamic form of attributes. Kiama's `DynamicAttribution` module defines attributes using the interfaces shown earlier and adds the `+=` operator to enable an attribute definition to be dynamically extended. Therefore, attribute grammar specifications can be separately compiled, dynamically loaded into the Java Virtual Machine, and added to an existing definition. This makes it possible to distribute language extensions in the form of binary plugins.

The extension operator is illustrated by Figure 7 that extends the dataflow example of Section 2.2 by adding a `Foreach` construct. The body of `DataFlowForeach` is a set of statements; the extension is only activated if these are executed. Each invocation of `+=` on an attribute adds a new definition to an internally maintained list of partial functions for the attribute. Inspired by the Disposable pattern [17], we introduce a method similar to the `using` statement in languages such as C#. With this technique, we can activate the extension as follows:

```
using (DataflowForeach) {
  ... // evaluate attributes using the extension
}
```

The extension is only active in the scope of the block of code, and any defini-

tions added are removed after it completes. The `using` method is implemented as follows: [6]

```
def using[T] (attributeInitializer : => AnyRef) (block : => T) =
    try {
        use (attributeInitializer)
        block
    } finally {
        endUse (attributeInitializer)
    }
```

That is, it uses two helper methods to first activate and keep track of new definitions, then evaluates the block, and finally removes the definitions again. Therefore, `using` allows extensions to be combined easily in a disciplined, scoped fashion.

5 Evaluation

We evaluate the performance of attribute evaluation in Kiama by a comparison to a handwritten Scala implementation and to a generated Java attribute evaluator. For the former, we specify attributes as regular methods in the AST classes, and perform caching by hand, at a cost of modularity and boilerplate code. For the latter, we compare to JastAdd, which, like Kiama, uses the Java platform and supports reference and circular attributes [11]. JastAdd has been successfully used to implement a full-featured Java 1.5 compiler that offers performance that can compete with handwritten implementations [9].

As a test case, we use the JastAdd example PicoJava specification from [27], which has 18 abstract syntax productions and 10 attributes to perform name and type analysis. We tested evaluation performance for relatively large, generated input programs. Since PicoJava only supports class definitions and not methods, our input classes contain 150 nested class definitions.

Figure 8 shows our benchmark results. The LOC column shows the number of lines of non-commented code to implement the AST and attribute grammar in each specification. The timings show the amount of time used for 100 runs evaluating the `errors` attribute that uses the other attributes to check for naming problems and cycles in the inheritance hierarchies. We first constructed a list of 100 inputs and evaluated the attribute for each input. In a second series of tests, we constructed only a single input in each run, ensuring that older inputs could be garbage collected, minimizing memory overhead of the benchmark. We used this approach because the input classes are particularly large for JastAdd which uses fields for caching. In both cases, we only timed the attribute evaluation process, ignoring input/output and tree creation overhead

The original JastAdd specification only used caching on selected attributes, which for our test cases appeared to lead to a decrease in performance. Thus, we created a variation where all attributes were cached, and finally a further variation that disabled JastAdd's use of rewrite rules. The Kiama implementation is a direct translation of this last variant. The results indicate that Kiama, while not using code generation and having had little performance optimisation, provides

[6] A parameter type `=> T` denotes a call-by-name parameter.

	LOC	*List of classes*	*Single classes*
Java: JastAdd	252	4010 ms	3459 ms
Java: JastAdd (full caching)	252	1767 ms	952 ms
Java: JastAdd (full caching, no rewrites)	243	1260 ms	860 ms
Scala: Kiama (full caching, no rewrites)	262	1889 ms	2435 ms
Scala: Handwritten	424	862 ms	543 ms

Fig. 8. Lines of non-commented code (LOC) in the benchmark specifications and times to evaluate the `errors` attribute of a large PicoJava input program.

competitive performance to JastAdd and adds only limited overhead to a handwritten specification. Of course, like JastAdd, Kiama offers superior modularity and a more concise notation than is possible with the handwritten implementation.

6 Discussion and Related Work

This section briefly compares the approach taken to develop the Kiama attribution library with generator-based systems that feature a dynamic evaluation approach. We are not aware of another attribute grammar system that uses pure embedding.

In many ways, Kiama has been inspired by the JastAdd [11] system and the features provided are similar. JastAdd provides an object-oriented variation of attribute grammars, supporting inheritance in their definition and references as attribute values [11]. Like JastAdd, Kiama is based on the Java platform, but makes use of the Scala language rather than a pre-processor approach.

The JastAdd approach to attribute evaluation might be characterised as “roll your own” laziness for Java. Scala does have lazy values, but to use them in Kiama would require attribute definitions to reside in the abstract syntax classes, which goes against modularity. Therefore we use the same general approach as JastAdd, but cache the values in attribute objects rather than in the tree nodes. This design implies some space overhead but we haven’t observed it to be a problem in practice.

A number of systems use lazy functional languages to define evaluators as circular programs [13]. The most prominent recent projects are LRC [23], Silver [25], the UU AG system [2] and first-class attribute grammars [5]. Built-in laziness means that explicit scheduling of attributes is avoided. Fully circular attributes are not possible by default but a form of circularity can be obtained [1]. In contrast, Kiama’s approach is more work to implement but more flexible because we have lightweight, fine-grained control over the mechanisms used to evaluate attributes while retaining the property that schedules are computed implicitly.

All of the systems cited use a special-purpose front-end or pre-processor to translate their attribute grammars into an implementation language, Java in the case of JastAdd and Haskell for the other systems. Since attribute equations in these systems are largely written in the syntax of the target language, a fairly high level of integration is achieved. Kiama removes the generator completely. While a custom input language is often desirable, the benefits can be outweighed by the simplicity and light-weight nature of an approach that doesn’t need a generator with its associated learning curve and influence on the build process. Scala’s expressive nature limits the sacrifices that must be made when making this tradeoff.

The first-class attribute grammars work [5] gets the closest to a pure embedding since attributes are first-class citizens that can be combined using combinator functions. As such, it shows similarities to our dynamically extensible attributes. However, the syntax used is supported by a pre-processor, rather than using pure Haskell. Based on the Haskell type checker, first-class attribute grammars prevent errors where the use of an attribute does not match its type. Errors due to cyclic dependencies or a mismatch between attribute equations and grammar productions are not reported. In earlier work, De Moor et al [6] used a Rémy-style record calculus to detect errors of the latter category, but this was found to be too restrictive.

One advantage of a generator-based approach is the ability to check the attribute grammar for correctness at generation time. For example, completeness and well-formedness checks [7] give confidence that the generated evaluator is not incomplete. In Kiama, precise checking of this kind is not always possible, particularly if syntax extensibility is desired. A Scala case class can be marked `sealed` which means that it cannot be extended outside the current module. When compiling a pattern match against a sealed class, the Scala compiler can emit warnings if the patterns are not complete, giving Kiama a form of completeness checking.

Kiama's encoding of the abstract syntax grammar in case classes also removes the possibility of some grammar-based checks. For example, in the Repmin example of Section 2.1 a run-time type check was necessary to ensure that the parent of a tree node was also a tree node. This check would not be necessary in a grammar-based generator, since the relationships between non-terminals could be determined statically. In practice, however, checks of this kind are not needed often and therefore do not outweigh the advantages of using standard Scala case classes as Kiama's tree representation.

As mentioned in Section 4, many attribute grammar systems allow the grammar to be written as separate "aspects" that are automatically "woven" together at generation time. In contrast, Kiama requires explicit descriptions of composition in the specification. While automatic composition of aspects is certainly convenient, in a general-purpose language setting where explicit composition is the norm its absence is not really felt.

7 Conclusion and Future Work

Dynamically-scheduled attribute grammars are a powerful language processing paradigm that has been the focus of many generator-based implementations. In most cases, a general purpose language is used to express attribute computations. The Kiama attribution library removes the generation step by using Scala to write the whole attribute grammar. The resulting system is lightweight and easy to understand, yet capable of competing in expressivity and performance with JastAdd, a mature generator-based system which uses a similar evaluation method.

The Scala features used by Kiama are present in one form or another in other languages, although usually not together. Scala's powerful expression language and first-class functions are well complemented by object-oriented features for state

encapsulation and modularity. More than any other feature, Kiama benefits most from Scala's pattern-matching anonymous functions that support the clean and natural attribute equation notation.

Kiama is in active development. For example, we are adding collection attributes [4,15]. Scala's ability to extend traits that define values (as opposed to methods) is being improved, which we plan to use to provide better support for defining a single attribute in multiple modules. The general question of analysis for embedded languages is also interesting for Kiama since it could lead to a solution that is both modular and provides better static completeness guarantees.

References

[1] Augusteijn, A., "Functional programming, program transformations and compiler construction," Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands (1993).

[2] Baars, A., D. Swierstra and A. Löh, *UU AG System User Manual*, Department of Computer Science, Utrecht University, September (2003).

[3] Bird, R., *Using circular programs to eliminate multiple traversals of data*, Acta Informatica **21** (1984), pp. 239–250.

[4] Boyland, J. T., *Remote attribute grammars*, J. ACM **52** (2005), pp. 627–687.

[5] de Moor, O., K. Backhouse and S. Swierstra, *First-class attribute grammars*, Informatica **24** (2000), pp. 329–341.

[6] de Moor, O., S. Peyton-Jones and E. Van Wyk, *Aspect-oriented compilers*, in: *Proceedings of International Symposium on Generative and Component-based Software Engineering*, LNCS **1799** (1999), pp. 121–133.

[7] Deransart, P., M. Jourdan and B. Lorho, "Attribute Grammars: Definitions, Systems and Bibliography," Lecture Notes in Computer Science **323**, Springer-Verlag, Berlin, Germany, 1988.

[8] Ekman, T. and G. Hedin, *Modular name analysis for Java using JastAdd*, in: *International Summer School in Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science **4143** (2006), pp. 422–436.

[9] Ekman, T. and G. Hedin, *The JastAdd extensible Java compiler*, in: *Proceedings of the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA'07)* (2007), pp. 1–18.

[10] Hedin, G., *Reference Attributed Grammars*, Informatica (Slovenia) **24** (2000), pp. 301–317.

[11] Hedin, G. and E. Magnusson, *JastAdd: an aspect-oriented compiler construction system*, Sci. Comput. Program. **47** (2003), pp. 37–58.

[12] Hudak, P., *Modular domain specific languages and tools*, in: *Proceedings of the 5th International Conference on Software Reuse* (1998), pp. 134–142.

[13] Johnsson, T., *Attribute grammars as a functional programming paradigm*, in: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1987), pp. 154–173.

[14] Jourdan, M., *An optimal-time recursive evaluator for attribute grammars*, in: *Proceedings of the International Symposium on Programming*, Springer, 1984, pp. 167–178.

[15] Magnusson, E., T. Ekman and G. Hedin, *Extending attribute grammars with collection attributes - evaluation and applications*, in: *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (2007).

[16] Magnusson, E. and G. Hedin, *Circular Reference Attributed Grammars-their Evaluation and Applications*, Electronic Notes in Theoretical Computer Science **82** (2003), pp. 532–554.

[17] Mariani, R., *Garbage collector basics and performance hints*, MSDN Library. http://msdn.microsoft.com/en-us/library/ms973837.aspx (2003).

[18] Mernik, M., J. Heering and A. M. Sloane, *When and how to develop domain-specific languages*, Computing Surveys **37** (2005), pp. 316–344.

[19] Nilsson-Nyman, E., G. Hedin, E. Magnusson and T. Ekman, *Declarative intraprocedural flow analysis of Java source code*, in: *Proceedings of the 8th Workshop on Language Descriptions, Tool and Applications*, 2008.

[20] Odersky, M., L. Spoon and B. Venners, "Programming in Scala," Artima Press, 2008.

[21] Odersky, M. and M. Zenger, *Scalable component abstractions*, in: *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications*, 2005, pp. 41–57.

[22] Paakki, J., *Attribute grammar paradigms—a high-level methodology in language implementation*, ACM Computing Surveys **27** (1995), pp. 196–255.

[23] Saraiva, J., "Purely Functional Implementation of Attribute Grammars," Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1999).

[24] Sloane, A. M., *Experiences with domain-specific language embedding in Scala*, in: *Proceedings of the 2nd International Workshop on Domain-Specific Program Development*, 2008.

[25] Van Wyk, E., D. Bodin, J. Gao and L. Krishnan, *Silver: an Extensible Attribute Grammar System*, Electronic Notes in Theoretical Computer Science (ENTCS) **203** (2008), pp. 103–116.

[26] Vogt, H. H., S. D. Swierstra and M. F. Kuiper, *Higher order attribute grammars*, in: *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (1989), pp. 131–145.

[27] *Picojava checker*, http://jastadd.cs.lth.se/examples/PicoJava/.