Science of Computer Programming ▌(▌▌▌▌) ▌▌▌–▌▌▌

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A pure embedding of attribute grammars[☆]

Anthony M. Sloane[a,*], Lennart C.L. Kats[b], Eelco Visser[b]

[a] *Department of Computing, Macquarie University, Sydney, Australia*

[b] *Software Engineering Research Group, Delft University of Technology, Delft, The Netherlands*

ARTICLE INFO

Article history:
Received 14 March 2010
Received in revised form 18 March 2011
Accepted 23 November 2011
Available online xxxx

Keywords:
Language processing
Compilers
Domain-specific languages

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 on demand. In this paper, we show how to remove the need for a generator, by embedding a dynamic approach in a modern, object-oriented and functional programming language. The result is a small, lightweight attribute grammar library that is part of our larger Kiama language processing library. Kiama's attribute grammar library supports a range of advanced features including cached, uncached, higher order, parameterised and circular attributes. Forwarding is available to modularise higher order attributes and decorators abstract away from the details of attribute value propagation. Kiama also implements new techniques for dynamic extension and variation of attribute equations. We use the Scala programming language because of its 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, traits and mixins for composition and implicit parameters for forwarding. A benchmarking exercise shows that our approach is practical for realistic language processing.

1. Introduction

The language processing domain is concerned with 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 [1,2].

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 tree traversal 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 [3]. LRC [4], JastAdd [5], UU AG [6], Silver [7], and first-class attribute grammars [8] are prominent examples of this approach. A dynamic schedule

[☆] Expanded and revised version of an article presented at the 9th Workshop on Language Descriptions, Tools and Applications (LDTA'09). This research was supported by The Netherlands 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*.

[*] Corresponding author. Tel.: +61 0298509582; fax: +61 0298509551.
E-mail addresses: Anthony.Sloane@mq.edu.au (A.M. Sloane), L.C.L.Kats@tudelft.nl (L.C.L. Kats), visser@acm.org (E. Visser).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.11.005

has the advantage that attributes are evaluated at most once, since attributes that are never needed will never be evaluated. However, keeping track of which attributes have been evaluated 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 development processes. We show in this paper how to integrate a dynamically scheduled attribute grammar approach as a library into an existing modern programming language. We use a *pure embedding* where the syntax, concepts, expressiveness and libraries of the host language are used directly [9,10]. The high-level declarative nature of the attribute grammar formalism is retained and augmented with the flexibility and familiarity of the host language, both for specification and for implementation of the formalism itself.

This work is part of the Kiama project [11] which is investigating pure embedding of language processing formalisms into the Scala programming language [12]. Kiama can be downloaded from http://kiama.googlecode.com, along with documentation and the full code for the examples presented in this paper. Kiama is not restricted to attribute grammars, but its other capabilities, such as strategy-based term rewriting, are beyond the scope of this paper.

Scala is an interesting and powerful host language for the Kiama project due to its inclusion of both object-oriented programming and functional programming features, emphasis on scalability and interoperability with the Java virtual machine. Many domain-specific languages have been implemented as Scala libraries, including various testing frameworks (notably ScalaTest [13] and Specs [14]) and ScalaQL [15] for interfacing to databases. Kiama particularly takes advantage of Scala's support for domain-specific languages via features such as higher order functions, general operator notations, flexible syntax [16] and implicit definitions of parameters [12].

Even though Kiama is a library, its attribution features have the same general power as generator-based systems such as JastAdd [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 providing basic synthesised and inherited attributes, Kiama currently supports various forms of higher order attribute [17,18], parameterised attributes [19], attribute forwarding between tree nodes [20], circular attributes that are evaluated to a fixed point [21], and abstractions for attribute value propagation patterns [22]. Language extension and modification are achieved using Scala's scalability constructs such as implicit parameters, traits and mixins [12]. 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 in the same ballpark as the dynamically scheduled evaluators produced by JastAdd when we compare performance on a limited case study.

The paper is structured as follows. First, in Section 2 we present Kiama's basic attribute grammar facilities, their implementation as functions with access to generic structural properties of the tree that is being attributed, and attribute value propagation using decorators. In Section 3, we consider the use of higher order attributes to implement cross-tree references and transformations, encountering tree splicing, parameterised attributes, and forwarding between trees along the way. Section 4 shows how circular attributes can be used to specify computations that execute to a fixed point and Section 5 discusses how Scala's component facilities enable attribute definitions to be modularised. Section 6 compares the performance of Kiama with JastAdd on an attribute grammar that is typical of those used to specify compilers. The paper concludes with a discussion of our approach in the context of other attribute grammar systems in Section 7 and concluding remarks in Section 8.

This paper is an extended and revised version of one that appeared in the proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA). Apart from extending the description in all sections, we have added new material describing additions to the library since the LDTA paper: attribute propagation patterns, transformation, tree splicing and forwarding.

2. Basic attribution

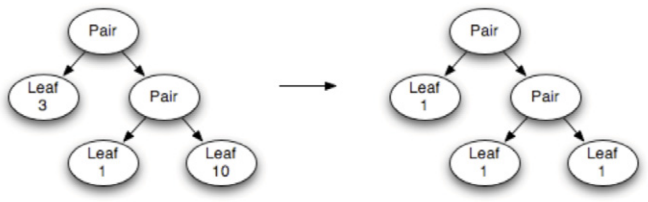
Attribute grammars augment a syntax definition with equations that define *attributes* of grammar symbols. An evaluator for a particular attribute grammar is able to traverse trees that conform to the syntax and compute the values of the attributes at each tree node. For example, a favourite basic example of attribute grammars is the *Repmin* problem which involves processing of binary trees with integers at the leaves [23]. The aim is to construct a new tree with the same structure as the input tree, but where each leaf value is replaced by the minimum leaf value of the entire tree. Fig. 1(a) shows a typical Repmin problem instance.

A typical attribute grammar specification for the Repmin problem is as follows[1]:

```
locmin : Int
    Pair (l, r).locmin = min (l.locmin, r.locmin)
    Leaf (v).locmin    = v

globmin : Int
```

[1] The notation used here is not the notation of any specific attribute grammar system, but is intended to illustrate the basic elements of any attribute grammar notation.



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

(a) A typical Repmin problem instance in graphical and Scala notation. The tree on the left is transformed into the tree on the right which has the same structure, but where each leaf value has been replaced by the minimum leaf value.

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

(b) Scala case classes that define the abstract syntax for attributable binary trees.

```
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)
    }
```

(c) Kiama attribute grammar for Repmin.

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

```
        t.globmin = if (t.isRoot) t.locmin else t.parent.globmin

    repmin : Tree
        Pair (l, r).repmin = Pair (l.repmin, r.repmin)
        t.repmin           = Leaf (t.globmin)
```

Associated with each attribute are equations that define the attribute's value for different types of tree nodes. The notation `n.a` accesses the `a` attribute of node `n`. The `locmin` and `globmin` attributes map a tree node to the minimum value in the subtree rooted at that node and the minimum value in the whole tree, respectively. The `repmin` attribute translates a node into its corresponding node in the result tree. `locmin` and `repmin` are called *synthesised attributes* because they rely on information from the node or its children. `globmin` is an *inherited attribute* because it is defined by information from its ancestors. It is common to have pre-defined attributes, such as `isRoot` and `parent` that provide reflective access to the tree structure.

The definition style used in this example is "attributes first" where the equations for each attribute are grouped together. This corresponds to the style used in Kiama. A "productions first" style is also common in attribute grammar systems, where all of the equations for a particular production are grouped together.

The main advantage of attribute grammars is that they separate the problem-specific equations from the details of attribute storage and tree traversal. The attribute grammar system or library is responsible for implementing an evaluator that obeys the attribute dependencies. In contrast, hand-coded approaches to tree traversal and decoration, such as the visitor design pattern, require the developer to manually ensure that these details are correct even as attribute definitions are changing during development.

2.1. Repmin in Kiama

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

Fig. 1(b) shows the abstract syntax for the Repmin binary tree structure in Scala. The `Tree` class inherits from Kiama's `Attributable` trait to obtain generic functionality (described below), but otherwise it is a regular Scala datatype definition. The case classes can have members, other supertypes and so on, without affecting the attribution.

Fig. 1(c) shows a Kiama solution to the Repmin problem.[2] *Attribute equations* are defined by cases that match on the tree node, wrapped in a call to Kiama's `attr` function. The normal pattern matching abilities of case classes can be used in the equations. 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`. Kiama does not use the more standard dot notation since this is already claimed by Scala for value and method references.

2.2. Structural properties

The definition of `globmin` uses generic *structural properties* to inspect the tree structure. In this example the properties used are `t isRoot`, which is true if `t` is the root of the tree, and `t.parent`, which is a reference to `t`'s parent. Note that since the parent has a generic type, it must be cast to a `Tree` in order to access its `globmin` attribute. Section 7 revisits this typing question.

Case classes to be attributed must inherit from the `Attributable` trait if structural properties are required. The `->` operator comes from `Attributable` as well. Each case class is automatically an instance of Scala's `Product` trait which provides generic access to its 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`.

The attribution library must coexist with Scala code that processes the same data structures. In particular, nodes might contain sequences and optional fields represented by Scala values of type `Seq[T]` (a sequence of values of type `T`) and `Option[T]` (an optional value of type `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, particularly primitive values.

Kiama makes a distinction between the *containment relation* connecting a node to its fields, which comes for free from the case class declaration, and the *parent–child relation* that relates an `Attributable` node to its `Attributable` children via its structural properties. Both of these relations are useful in attribute equations.

To illustrate the difference between these two relations, Fig. 2 shows an example where an `Upper` node contains four fields: one required `Lower` node, a sequence of zero or more `Lower` nodes, an integer and an optional `Lower` node. The `Upper` node therefore has five `Attributable` children and those nodes all have the `Upper` node as their `parent`. In other words, when a `Lower` child asks for its parent, it is more useful to get the `Upper` node, which has a semantic role in the tree structure, rather than the container in which it is stored. 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.

2.3. Attributes as functions

Attributes defined by `attr`, such as those used for our Repmin example, are implemented by the `CachedAttribute` class (Fig. 3). Since attribute values are not evaluated until they are needed and they are cached, the evaluation method is equivalent to those used in some early attribute grammar systems [3] and, more recently, in JastAdd [5].

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 constructor parameter `f` is the user-specified function that defines the attribute equations. A set of pattern matching cases are regarded by Scala as an anonymous function so it can be passed to this parameter as shown in the Repmin example. Alternatively, the attribute equations can be defined by a separate named function.

`CachedAttribute` is a sub-class of the function type `T => U` so it must define the method `apply`, that "runs" the function by using the defining equations on the given node, and returns the value of the attribute.

[2] Scala uses the notation `T => U` for total functions and `PartialFunction[T,U]` for partial functions. `T ==> U` is a Kiama type synonym for `PartialFunction[T,U]`. A function can be created by a brace-delimited expression consisting of a sequence of cases. Each case consists of a pattern match, then =>, then the result expression of the case. Within a pattern, identifiers beginning with a lowercase letter are binding occurrences, whereas those beginning with an uppercase letter are constants. An underscore pattern matches anything. A pattern `v @ p` binds the name `v` to the value matched by the pattern `p`. A guard `if boolexp` allows the associated case to match if the expression `boolexp` evaluates to true. Scala allows the period in a method call to be omitted in some situations, so `t isRoot` is just `t.isRoot`, `n->a` is just `n.->(a)`, and similarly for the call of the `min` method in the definition of `locmin`.

```
case class Upper (a : Lower, b : Lower*, c : Int,
                  d : Option[Lower]) extends Attributable
case class Lower (...) extends Attributable
```

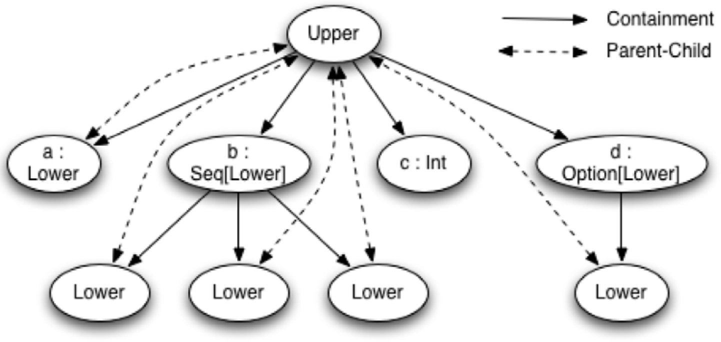


Fig. 2. The Kiama parent–child relation compared to structure containment. The asterisk annotation on the type of the b field denotes a repeated value, which is passed as a sequence.

```
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      => error ("Cycle detected")
            case Some (u) => u
            case _         => memo.put (t, None)
                              val u = f (t)
                              memo.put (t, Some (u))
                              u
        }

}
```

Fig. 3. Basic attributes are implemented by the `CachedAttribute[T,U]` class which adds caching behaviour to the function type `T => U`.

Scala converts `a(n)` into `a.apply(n)`, so the implementation of attributes as functions provides a convenient default notation for access to the attribute value. For better compatibility with other attribute grammar notations, the reference-style notation `n->a` is provided by `Attributable` as an alias for `a(n)`. The implementation of `->` is as follows, where `this` is the node on which the attribute is being referenced and `this.type` is its type.

```
def ->[U] (a : this.type => U) = a (this)
```

Given the attribute definition a–a function from the node type to the attribute type U—we simply apply a to the node to get the attribute value.

In `CachedAttribute` a local memo table is used to keep track of the attribute values. When the attribute's value is first demanded, `memo.get` returns `null` (third case), indicating that a value is not yet available for this attribute. A `None` value is inserted into the memo table to indicate that the attribute is being evaluated, then the attribute definition (`f`) is called. The result of the evaluation replaces the `None` in the map. If the attribute's value is sought during its own evaluation (first case), an error is raised. (See Section 4 for equations that deliberately involve a cycle.) Otherwise (second case), the value from the memo table is returned when the attribute is demanded after the first time.

2.4. *Attribute propagation patterns*

The Repmin solution in Fig. 1(c) works, but may be more verbose than desired. In particular, it uses a common attribute propagation pattern to define the `globmin` attribute: defining an attribute at the root of the tree and passing it down to where it is needed. Ideally, we would like to remove this kind of "boilerplate" so that the attribute propagation patterns can be reused and the equations can be limited to the problem-specific cases.

In previous work, we introduced the idea of using *attribute decorators* to express patterns of attribute value propagation [22]. The idea is that generic access to the tree structure enables propagation patterns to be factored out and

```
val globmin : Tree => Int =
    down {
        case t if t isRoot => t->locmin
    }
```

(a) Defining the globmin attribute using the down attribute propagation pattern.

```
def down[T <: Attributable,U] (a : T ==> U) : T => U =
    attr[T,U] {
        case t =>
            if (a.isDefinedAt (t))
                a (t)
            else
                (down (a)) (t.parent[T])
    }
```

(b) The down attribute propagation pattern.

Fig. 4. Removing boilerplate from the Repmin attribute grammar.

reused. In that work, we used decorators implemented by generic tree traversal strategies to demonstrate this idea in the Aster system which is based on Stratego [24].

Fig. 4(a) shows how the `globmin` attribute equations can be simplified using a value propagation pattern similar to a decorator. We use a `down` pattern which propagates a value down the tree, given a definition of the attribute at an ancestor node (in this case, the root). `down` is defined using the `parent` structural property (Fig. 4(b)).[3] First, the parameter partial function `a` is queried using its `isDefinedAt` method to see if it provides a definition for the node `t`. If it does, then `a` is applied; otherwise, the `down(a)` attribute is invoked recursively on the parent of `t`.

3. Higher order attributes

The `locmin` and `globmin` attributes from the previous section are first order because they map tree nodes to non-tree values. In contrast, the `repmin` attribute is a *higher order attribute* because its value is itself a tree. The value of the `repmin` attribute is used only to produce the output structure, but higher order attributes are more powerful in general because they can themselves have attributes.

The literature distinguishes between higher order attributes whose values are new tree structures [17], and those that reference existing nodes in the tree that is being attributed. The latter kind are usually accompanied by a change from value semantics for attributes to reference semantics, so they are called *reference attributes* [18,21].

Kiama supports both of these kinds of attributes by allowing the value of an attribute to be a reference to a tree node. It can be a reference to an existing node or to a new one. No new evaluation machinery is needed because, under a dynamically scheduled evaluation scheme, asking for an attribute of a reference forces the reference to be evaluated first.

3.1. Control flow

As a first example of higher order attributes, consider computation of control flow information for an imperative programming language. One way to formulate this problem as an attribute grammar is to express the control flow via a reference attribute `succ` that gives the set of statements that can follow a given statement in a program execution. The statements will be represented by references to their tree nodes, effectively superimposing a graph structure on the tree.

Fig. 5(a) gives the abstract syntax of a simple imperative language containing assignment, while, conditional, block, return and empty statements. Fig. 5(b) shows the Kiama definitions of the control flow attribute.[4] The control flow successor `succ` is a synthesised reference attribute defined in terms of a `following` attribute that defines the default linear control flow. For example, the successor of a node `t` representing a while loop is the union of the statements that follow the while loop (`t->following`) and the statement that is the body of the loop (`s`). `following` is defined as an inherited attribute by pattern matching on first the current node (outer case), and then on the parent node (inner cases), using the convenience function `childAttr` which is defined by the following equivalence:

```
childAttr(f) == attr { case t => f(t)(t.parent) }
```

[3] A Scala generic type declaration can include upper and/or lower bounds. `T <: Attributable` in the definition of `down` means that the type substituted for `T` must be a sub-type of `Attributable`. This constraint guarantees that the `parent` property is available.

[4] The `_*` patterns in these definitions are wildcards that match possibly-empty sequences.

```
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
```

(a) Abstract syntax for a simple imperative language.

```
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 ()
        }
    }
```

(b) Attribute definitions for control flow. `childAttr` is similar to `attr`, except that its argument function matches first on the current node (the child) and then on its parent.

Fig. 5. Computing control flow for an imperative language.

3.2. Transformation

Higher order attributes that refer to new tree nodes can be used to specify transformations. Consider an expression language that has user-defined operator priority (based on the example in Vogt et al. [17]). A standard way to process such a language is to parse an input expression into a tree that disregards operator priority, in our case a right-recursive tree, then perform analysis of the input tree to compute a new tree that has a structure that takes priority into account. Fig. 6(a) shows an example from [17], assuming that operators are left associative and that multiplication has a higher priority than addition.

To define a Kiama solution for this problem, we first need to specify the tree structures. Fig. 6(b) gives the Scala definitions for a right recursive tree structure (`ExpR`) and a general binary expression tree structure (`Exp`). To simplify the transformation, the leaf node types `Num` and `Var` are shared between the two structures.

Fig. 7 shows Kiama attribute definitions to solve this transformation problem using the algorithm presented in Section 2.6 of [17].[5] If `r` is a right-recursive tree, then `r->op_tree` is the equivalent operator tree with priorities resolved. The algorithm operates by moving down the right-recursive tree, maintaining stacks of operators and operands (calculated by the `ops` attribute), until a priority difference is found. The auxiliary function `eval_top` makes these decisions and creates the new tree. `prioenv` is assumed to map operators to their priorities.

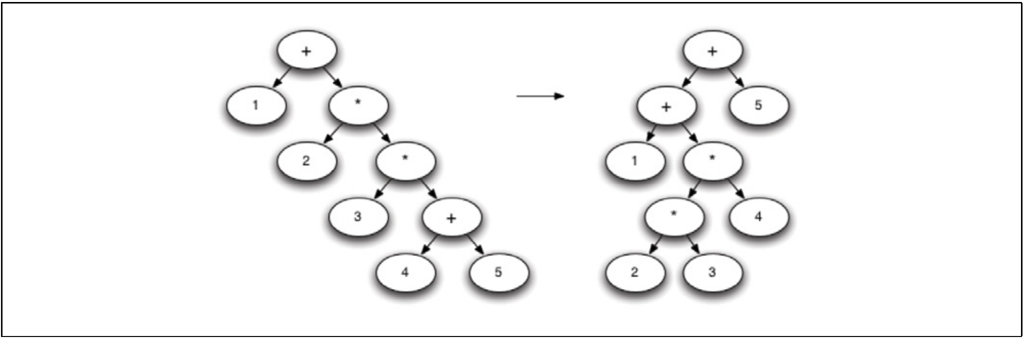
3.3. Tree splicing

Suppose that we wish to perform name or type analysis on the priority-corrected trees of Section 3.2. Assume that each expression in the original tree is embedded in a context that provides definitions for names via some scoping structure. As it is, we cannot directly perform name analysis on the `op_tree` attribute of an expression, because there is no connection back from the `op_tree` to the original tree that contains the name declarations.

In a traditional attribute grammar system such as that described in [17], this connection is achieved by an attribute rule such as the following.

```
R → R1 R2
    R2 := R1.op_tree
```

[5] Scala lists are constructed from `Nil` and the "cons" method `::`. A pattern of the form `p : t` matches only if the value matched by `p` is of type `t`.



(a) Transformation of a right-recursive expression to one that takes operator priority into account.

```
abstract class ExpR extends Attributable
case class BinExpR (left : Exp, op: String, right : ExpR)
    extends ExpR
case class Factor (exp : PrimExp) extends ExpR

abstract class Exp extends Attributable
case class BinExp (left : Exp, op: String, right : Exp)
    extends Exp

abstract class PrimExp extends Exp
case class Num (value : Int) extends PrimExp
case class Var (name : String) extends PrimExp
```

(b) Abstract syntax for right-recursive expressions (ExpR) and general binary expressions (Exp).

Fig. 6. Transformation to resolve binary operator priorities.

R1 is the original right recursive tree and R2 is a so-called *non-terminal attribute* whose value is the priority-corrected tree. The equation has the effect of *splicing* the new tree into the old tree as the second child of the R node. The system must ensure that no attributes of R2 are accessed until this splicing operation has occurred.

An analogous effect can be achieved in Kiama using the `tree` attribute definition function. An attribute defined with `tree` must have a tree value. If `a` is such an attribute, then evaluating `n->a` evaluates `a` as if it were defined using `attr` *and* sets the parent of `n->a` to `n.parent`. Thus, the new tree gains a context that is the same as the existing node.

We do not want to splice every `op_tree` attribute into the old tree, only at the top of each nested expression. Hence, we define an `ast` attribute that has the same value as `op_tree` but is a tree attribute. We can then evaluate `n->ast` at exactly those places where a splice should occur and we are guaranteed that the context has been set correctly.

```
val ast : ExpR => Exp =
    tree {
        case e => e->op_tree
    }
```

3.4. Name analysis, parameterised attributes and error handling

A common application of reference attributes is name analysis for compilers and other software analysis tools. The idea is that instead of representing name information in a separate environment, it is stored in existing tree nodes and references to those nodes replace references to environment entries [18].

Using this approach, we can define name analysis on the priority-corrected tree using reference attributes as follows. We define a *parameterised attribute* `lookup` that can be used at any node to look for the declaration of a name (Fig. 8(a)). Parameterised attributes in Kiama are defined using `paramAttr` which manages caching of the attribute value using a key that depends on both the parameter value and the node whose attribute is being evaluated.

In the definition of `lookup`, `s` is the name being looked up and the inner cases match against the node at which we are looking. The equations search up the tree until reaching the root, which we assume is a `Program` node that has a `vars` field containing a list of the variable declarations. (This approach can be extended in obvious ways to deal with nested scopes and so on.) At the root we perform a search in the variable declarations for one that declares the name in which we are interested and return it wrapped as an `Option`. If it's not there, we return `None`.

A simple error checking attribute can be defined in terms of `lookup` (Fig. 8(b)). At `Var` nodes we use `lookup` to check that the name used is declared. Clearly, the declaration, once obtained, could also be used to access type information and other variable properties.

```
val op_tree : ExpR => Exp =
    attr {
        case BinExpR (_, _, e1) => e1->op_tree
        case e1 @ Factor (e)    =>
            val (optor, opnd) = e1->ops
            val (_, es) = eval_top (optor, null, e :: opnd)
            es.head
    }

type Stacks = (List[String], List[Exp])

val ops : ExpR => Stacks =
    childAttr {
        case e1 => {
            case _ : Program              => (Nil, Nil)
            case e0 @ BinExpR (e, op, _) =>
                val (optor, opnd) = e0->ops
                eval_top (optor, op, e :: opnd)
        }
    }

def eval_top (optor : List[String], op : String,
              opnd : List[Exp]) : Stacks =
    optor match {
        case Nil                => (List (op), opnd)
        case top_op :: rest_ops =>
            if (prioenv (top_op) < prioenv (op))
                (op :: top_op :: rest_ops, opnd)
            else {
                val o1 :: o2 :: rest = opnd
                eval_top (rest_ops, op,
                          BinExp (o2, top_op, o1) :: rest)
            }
    }
```

Fig. 7. Attribute definitions to resolve binary operator priorities.

```
val lookup : String => Attributable => Option[VarDecl] =
    paramAttr {
        s => {
            case p : Program => p.vars.find (_.name == s)
            case e           => (e.parent)->lookup (s)
        }
    }
```

(a) A parameterised lookup attribute for resolving names. The `paramAttr` function takes an argument that first matches on the parameter of the attribute and then on the node whose attribute is sought.

```
val errors : Exp => Unit =
    attr {
        case BinExp (l, o, r) =>
            l->errors; r->errors
        case e @ Var (s) if (e->lookup (s) == None) =>
            message (e, s + " is not declared")
        case _ =>
            // no error
    }
```

(b) Collecting errors from name analysis.

Fig. 8. Name analysis.

The `errors` attribute adds to a Kiama global message store using the `message` function which takes the relevant node (for coordinate information) and the message. This approach is the easiest way to collect messages. It is also possible to use a pure attribute approach by defining `errors` to evaluate to a set of messages that are passed up the tree for reporting at the root.

With all of these attributes in place, given a right-recursive expression n, we can generate the equivalent priority-corrected tree and check it, using the expression

```
n->ast->errors
```

Evaluating ast forces the op_tree to be built and spliced in, providing the context in which the errors attribute can use lookup correctly.

3.5. Attribute forwarding

A problem with the formulation of the previous section is that we need to know that the n->ast higher order attribute provides the errors attribute for a right recursive tree n. Exposing the ast attribute to clients reveals details about the transformation attribution that ideally would only be known to the transformation module.

Attribute grammars that are more modular can be written using a technique called *attribute forwarding* [7,20]. A forwarding declaration specifies another node to which a request for an attribute a should be sent if the receiving node does not have a definition of a. Usually the declaration is associated with a node type, so all nodes of that type forward in the same way. Thus, in our example, our transformation module can specify that a right recursive tree forwards to its ast attribute. Attributes such as n->ast->errors can then be accessed by client modules directly as n->errors and the ast attribute does not have to be used directly.

Kiama implements forwarding via Scala's *implicit parameter* mechanism. The actual value of an implicit parameter can be omitted in a method call, provided that a unique implicit value of the appropriate type is in scope at the call. Values that are intended to be used as implicit parameter values must be marked with the implicit keyword.

As we saw in Section 2.3, the basic implementation of n->a simply applies the function a to the node n. -> is overloaded as follows to support forwarding.

```
def ->[T,U] (a : T => U) (implicit b : this.type => T) =
    a (b (this))
```

The attribute (parameter a) in this version is defined on some other type T. The implicit parameter b provides a way to get from the node type to T. b can be any function, but for forwarding it is usually a higher order attribute. The operator just applies b and then a to obtain the attribute value.

For n->errors to work, we need an implicit value of type ExpR => Exp, since n is an ExpR but the errors attribute is defined on Exp values. We use the ast attribute for this conversion, by simply marking it as an implicit value; the rest of its definition stays the same.

```
implicit val ast : ExpR => Exp = ...
```

The implicit definition of ast is exported by the transformation module so that it applies in any client modules. This, saying n->errors in a client, is the same as saying n->ast->errors. (The implicit modifier still allows the value to be used as a normal attribute; n->ast->errors remains a valid expression.)

This style of attribute forwarding is type-directed, which seems appropriate for a strongly-typed host language. When using forwarding in an implementation of a typical source-to-source transformation such as a desugaring rule, the types of the before and after trees must be different for the implicit version of -> to trigger. As we have seen in this example, this is not a limitation, since the interior structure of the two trees can share constructs. Only the type of the root of the new tree must be different from the type of the node whose attribute is being computed.

4. Circular attributes

The attribute equations we have considered so far are illegal if they define an attribute value to depend on itself. Kiama's dynamic scheduling approach means that these cycles are detected at run-time. In some cases, however, it is desirable to have an attribute that is defined in such a cyclic fashion. Typically, these are attributes whose definition is really the fixed point of some computation. Kiama can define such attributes via a simple extension of the basic attr function.

Consider a variable liveness computation for the imperative language used in the control flow example in Section 3.1 (adapted from [25]). Fig. 9(a) shows a typical variable liveness problem instance for this language, where the *in* and *out* sets are the live variables reaching and leaving each statement, respectively. For example, variable x is live into the while loop because it is defined in the previous statement and used in the loop condition.

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

These computations can be defined in a natural way in Kiama using the circular attribute definition function. Fig. 9(b) shows the relevant definitions. circular is like attr, except that it also takes an initial value for the attribute (here, an empty set of strings, denoted as Set[String]()) and evaluates until a fixed point is reached. We use the alternative attribute access notation a(n) for the liveness sets to emphasise the correspondence with the data flow equations. In the definition of *out*, note that the succ attribute defined in Section 3.1 is used to access the control flow. The Scala library method flatMap applies *in* to each of the statement's successors and combines the results (in this case using set union).

```
                      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.

```
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)
    }
```

(b) Attribute definitions to compute variable uses, definitions and liveness sets.

Fig. 9. Computing variable liveness for an imperative language.

`circular` is defined using a `CircularAttribute` class that provides a functional interface to the fixed-point evaluation algorithms of [21].

5. 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. For example, considering recent systems, the LIGA attribute grammar tool used in the Eli system [26] allows equations to be specified in different files, grouping by grammar production or attribute depending on developer preference. LIGA combines the equations before analysing the whole attribute grammar. LIGA also has notational support for modular reuse of equations [27]. JastAdd [5] uses an aspect-oriented approach where different attribution aspects are woven together at generation time. Silver [7] has a full module system for extensions, plus modularity support via forwarding. In contrast, the first-class attribute grammars approach [8] has attributes that are *first-class citizens* and can be manipulated directly as values.

While other attribute grammar systems often use a general-purpose language for the expressions in attribute equations (e.g., Haskell in UU AG [6], Java in JastAdd [5]), 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. Kiama's implementation of attributes as functions also opens up some opportunities for flexible composition.

5.1. *Static separation of concerns using traits*

Flexible static composition of attribution modules can be achieved using Scala traits to define components and performing mixin composition to combine them [28]. For example, we can decompose the variable liveness problem of Section 4 into three components: control flow (from Section 3.1), variable sets, 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* [28] 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 =>
    ...
    ... 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.

5.2. *Static separation of concerns using type-specific functions*

Traits and mixin composition support a form of modularity where different attributes are defined in different modules and are mixed together to form a complete solution. This approach implies that each attribute definition must deal with all possible syntactic constructs. It may also be desirable to decompose an attribute definition into parts that each deal with some feature of the language that is being processed. For example, a type checking attribute might be decomposed into type checking for expressions and for statements.

Since Kiama's attributes are just functions, they can be composed as functions. In particular, they can be partial functions, so it is open for an attribute designer to write separate equations for different syntactic types, then "chain" these equations together to form a single view of the attribute. For example, we might do the following to separate type checking equations, where we assume that the equations side-effect the global message store rather than returning a value.

```
val expcheck : Type ==> Unit = ...
val stmcheck : Stm ==> Unit = ...

val typecheck = expcheck orElse stmcheck
```

`orElse` is a method on partial functions that applies its argument function if the receiver is not defined.

This approach provides a great deal of flexibility at the price of requiring explicit composition of the equations. Also, a runtime cost is paid due to the separate pattern matching of the composed functions, in contrast to a single pattern match over all expected types if all of the equations were defined in one place.

5.3. *Dynamically extensible attribute definitions*

Kiama's use of functions to implement attributes leads to opportunities for even more flexible forms of composition. In this subsection we illustrate this flexibility by sketching the implementation of a new dynamic form of attribute. 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 Fig. 10 that extends the control flow definition of Section 3.1 by adding a `Foreach` looping construct. The body of the `DataFlowForeach` object is a set of statements; the extension is only activated if these are executed. Each invocation of += on a dynamic attribute adds a new definition to an internally maintained list of partial

```
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. 10. Dynamic attribute grammar extension.

functions for the attribute. Inspired by the Disposable pattern [29], 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 definitions added by it are removed after the `using` block completes.

5.4. *Productions first style*

Kiama naturally supports an "attributes first" style where all of the equations for an attribute are defined together. If independence between the syntax and attribute definitions is not desired, it is possible in Kiama to define attributes within the syntax classes to achieve a "productions first" style. In Scala it is not possible to directly add to class definitions after they have been defined. A language that has open classes, such as Ruby, would enable this style. In Scala an alternative is the so-called "pimp my library" pattern, where an implicit conversion is used to add methods to an existing type [30]. Attributes can be added to existing classes using this pattern.

6. Evaluation

Attribute grammar systems have traditionally used static scheduling of attribute evaluation, based on an analysis of a complete attribute specification in a specialised language. Recent systems have focused on dynamic scheduling of attributes, but are still based on compilation or interpretation of a specialised attribute grammar language. In contrast, Kiama takes a pure embedding approach that eliminates the interpretation or generation step.

A price must be paid for not having a specialised generation step, since domain-specific optimisations are hard or impossible to do. A comprehensive evaluation requires a full implementation of a non-trivial language that can be compared with other implementations. Such a comparison is prohibitively expensive, however, so we settle here for a more limited evaluation. Our aim is to see whether Kiama's embedding approach is practical when compared to a generation approach where there is more opportunity to tune the output. We compare the pure embedding approach of Kiama to the generative approach taken by the JastAdd system [5].

At run-time, JastAdd programs are evaluated much like Kiama programs, as both use dynamically scheduled attribute evaluation and are based on reference attributes. Both systems run on the Java Virtual Machine, ensuring that the basic operations have the same performance characteristics. JastAdd provides competitive performance: in the past it has been used to implement a full-featured Java 1.5 compiler with performance that could compete with handwritten compiler implementations [31].

We use the example PicoJava specification from the JastAdd website [32] for our benchmark. It uses 18 abstract syntax productions and 10 attributes to perform name and type analysis. Translation of the JastAdd specification to Kiama was straightforward and resulted in a program that followed the same structure and was virtually of the same size. We tested the performance of the PicoJava analyzer for relatively large, generated input programs that each consist of 150 class definitions.[6] This specification and these inputs are not necessarily representative of all language processing tasks and do

[6] The small PicoJava language only supports classes, not methods. To create an interesting test input we use inner classes to create a larger input program.

	LOC	*Time*
Java: JastAdd	252	3459 ms
Java: JastAdd (full caching)	252	952 ms
Java: JastAdd (full caching, no rewrites)	243	860 ms
Scala: Kiama (full caching, no rewrites)	262	2435 ms
Scala: Handwritten	424	543 ms

Fig. 11. Benchmark results showing lines of non-commented code (LOC) in the benchmark specifications and times to evaluate the `errors` attribute of a large PicoJava input program.

not exercise all of the features of Kiama and JastAdd (e.g., circular attributes), but they do give a basis for a simple direct comparison.

Fig. 11 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 of the PicoJava analyzer (invoking the `errors` attribute). We ran the tests on a 2.4 GHz machine, with a "warmed up" 32-bit Java 6 server JVM, creating a new tree before each run. We used Scala version 2.7.

A key factor in the performance of dynamically scheduled attribute grammar systems is attribute value caching. Attribute values that are cached may be reused for subsequent computations. In JastAdd, users can select which attributes should be cached using the `lazy` attribute. In Kiama, `attr` and its related functions provide caching; an uncached variant is available if necessary. The implementation of this concept is different in the two systems, however, which appears to account for a large part of the performance difference between the two. JastAdd generates Java classes that use fields to store cached attributes values in each AST node. Kiama uses per-attribute hash tables instead. Other, smaller performance differences between the systems may be expected as Kiama is based on Scala instead of Java.

To study the effects of the attribute caching mechanism, we also compared against a handwritten Scala implementation of PicoJava that uses fields instead of hash tables for caching. In this program, we did not use any Kiama facilities, except for support for structural properties of the abstract syntax tree. We wrote all attribute definitions as regular methods directly inside the AST classes and cached all attributes. As such, this implementation closely resembles the JastAdd-generated implementation with full caching but is written in Scala instead of Java. Thus, this implementation is a "productions first" style and, as such, gives up separation of syntax and attribute definitions.

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, which are not required for PicoJava. The Kiama implementation is a direct translation of this last variant. The results indicate that, for this specification, Kiama is quite a bit slower than the mature JastAdd system when full caching is used. Preliminary profiling shows that much of the overhead is due to the hash table implementation of caching that Kiama uses. The overhead is not prohibitive since Kiama evaluators run in reasonable time, particularly when considering that specialised code generation is not used and we have made no performance optimisations. However, we will be aiming to reduce this overhead in future implementations, particularly by taking advantage of the more mature collection library in current Scala versions.

The hand-written Scala specification is much more competitive, which is to be expected since the overhead of per-attribute hash tables is non-trivial. We expect that examination of performance bottlenecks in the future will reduce this overhead while retaining the modularity of the "attributes first" approach.

7. Discussion and related work

In this section we compare the approach taken to develop the Kiama attribution library with other attribute grammar systems that feature a dynamic evaluation approach.

The core implementation of attribution in Kiama consists of a few hundred lines of Scala code. We summarise the interface of the implementation in Fig. 12. Central to the implementation of Kiama is the `Attributable` type. It defines structural attributes of nodes that can be used in attribute equations. Equations themselves are based on methods of the `Attribution` singleton object that can be used from any type. As Kiama is fully defined in Scala, users can define their own variations of attribute definition functions to augment the cached, uncached and dynamically-extensible versions.

7.1. *Generator and pre-processor-based systems*

In many ways, Kiama has been inspired by the JastAdd [5] 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 [5]. Like JastAdd, Kiama is based on the Java platform, but makes use of the Scala language.

JastAdd implements a "whole-program" pre-processing of the attribute grammar syntax and equations. Separate modules are woven together at generation time and combined into a single body of generated Java code. Attribute definitions are implemented as methods in the abstract syntax classes. Thus, JastAdd allows modular specification but combines all of the modules together in the final program. This approach affects the scalability of specifications, since any change to an

`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`.
`tree (f : T => U) : T => U`	Higher order attribute defined by `f`.
`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.
`circular (init : U) (f : T => U) : T => U`	Circular attribute defined by `f` with initial value `init`.
Access attribute `a` *of node* `n`	
`n->a`	Reference style.
`a (n)`	Functional style.

Fig. 12. Summary of the Kiama attribution interface. Some of the functions impose restrictions on the types at which they can be applied (not shown). For example, `tree` can only be applied to types that extend `Attributable` since it must be able to connect the new tree to the old.

attribute definition requires the whole program to be re-generated and compiled. In contrast, Kiama's separation of syntax from attribute definition allows modular specification and scalable separate compilation.

The JastAdd approach to attribute evaluation might be characterised as "roll your own" laziness for Java: the JastAdd pre-processor generates boilerplate Java code that enables caching of values in the syntax classes. Scala does have lazy values, but to define an attribute as a lazy value of a particular syntax node, we would need to place the attribute definition in the relevant abstract syntax class. This requirement would negate the modularity and scalability benefits of separating the syntax from the attribute definitions. 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 [33]. The most prominent recent projects are LRC [4], the UU Attribute Grammar system [6], Silver [7], and first-class attribute grammars [8]. 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 [34]. In contrast, Kiama's approach is more work to implement than if laziness were built-in, but is 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.

Silver has been implemented in itself and has been designed to allow for modular extensions of the Silver language. Van Wyk et al. [7] demonstrated how automatic copy rules as well as more advanced features can be implemented in this fashion. While adding extensions of this kind is made easier through facilities such as forwarding for local transformations [20] and higher order attributes, it is hard to imagine an ordinary Silver user building such an extension. Moreover, it is difficult to encapsulate these extensions in a single application or library, as they must be integrated in the base attribute grammar system. In contrast, Kiama is implemented as a Scala library and can be extended directly within a Kiama program or as part of an additional library.

The first-class attribute grammars project [8] is close to a pure embedding since attributes are first-class citizens that can be combined using combinator functions. As such, they have similarities to our dynamically extensible attributes. However, the syntax used is supported by a pre-processor, rather than being 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. [35] used a Rémy-style record calculus to detect errors of the latter category, but this was found to be too restrictive.

In previous work, we described the Aster attribute grammar system and introduced the idea of using *attribute decorators* to express patterns of attribute value propagation [22]. Aster is a dynamically scheduled attribute grammar system based on the Stratego program transformation language [24]. Decorators are a form of higher order functions that allow the small Aster language to define a library with common attribute propagation patterns such as downward or upward copying of values, as well as more complicated patterns such as fixpoint iteration for circular attributes. In many ways these decorators correspond to the attribute definition functions of Kiama, although in Kiama they are based on the general-purpose Scala language rather than the generator-based Aster language.

In the present paper we focused on general-purpose attribute propagation patterns as they are found in various attribute grammar systems. In the Aster project we also described decorators for problem-specific patterns such as name analysis or error reporting [22]. These can be defined using structural attributes, strategic programming techniques [36], and by combining multiple decorators together for reuse. For the most part, similar patterns can be defined in Kiama, although Scala lacks the lightweight syntax for combining attribute definition functions (always requiring the `case x =>` construct) and does not have the reflective capabilities of the specialised Aster language. Then again, Kiama benefits from Scala's static type system, while Stratego is largely untyped. For example, programs that use Kiama can't create tree structures that don't conform to the abstract syntax, whereas Stratego programs can create any term; a separate step is needed to check grammar conformance.

7.2. Pure embeddings

All of the systems we discussed in the previous section 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 implementation language, a fairly high level of integration is achieved between attribute grammar specifications and non-attribute grammar code in the language. Kiama removes the generator completely. While a custom input language is often desirable, the benefits can be outweighed by the simplicity and lightweight nature of an approach that doesn't need a generator with its associated learning curve and influence on the build process. Scala's flexibility limits the sacrifices that must be made when making this tradeoff.

For example, Kiama's use of implicit parameters for forwarding leads to a more general feature with little implementation effort. A forwarding declaration in Van Wyk's Silver system [7] is provided in one place, can only forward to a node of the same type, and cannot be modified by other grammar aspects. In Kiama, different implicit values can be used to achieve different effects, either in different modules, or in the same module applying to different node types or different attributes. The scoped nature of implicit values requires that these values are used in a disciplined fashion. Also, Kiama separates the forwarding mechanism from that used to (optionally) splice the forwarded-to tree into the original tree, whereas Silver's declarations include both of these aspects. We believe our design separates concerns more appropriately, but we are still in the early days of exploring the extra flexibility.

Taking an embedded approach allows for better interoperability not just with existing general-purpose Scala code, but also with other languages embedded into Scala. For example, Scala's packrat parser combinators can be safely used together with Kiama. So can third-party embedded languages such as languages such as ScalaTest [13], Specs [14], and ScalaQL [15]. This would not be the case if both systems had their own pre-processor: a pre-processor for attribute grammars cannot easily be composed with one for another language extension.

Recent work by Viera et al. has shown how to improve on the first-class attribute grammars approach to embed attribute grammars into Haskell in a more satisfactory manner [37]. Conceptually this work is similar to ours in that a pure embedding approach is used. However, the complexity of the embedding is significant, involving heterogeneous lists and a non-trivial use of type classes and type-level programming. Their achievement is impressive from a technical viewpoint, but the extra complexity obscures what is going on considerably. In contrast, equations embedded using Kiama's approach closely resemble the notations one would use in an attribute grammar system with a custom specification language and Kiama's implementation is simple. Also, there is no need for complex types to define simple attributes or their propagation patterns.

One advantage of a generator-based approach is the ability to check the attribute grammar for correctness at generation time. For example, checking for completeness and well-formedness [1] gives 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, 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 except for parent references, since Scala knows the types of children. Overall, losing some grammar knowledge is unfortunate, but it is a small price to pay for the convenience and familiarity of using standard Scala case classes for Kiama's tree representation.

As mentioned in Section 5, many attribute grammar systems allow the grammar to be written as separate "aspects" that are automatically "woven" together at generation time. While automatic composition of aspects is certainly convenient, Kiama requires explicit composition in the specification, since that is in keeping with Scala's component features.

8. Conclusion and future work

Dynamically-scheduled attribute grammars are a powerful paradigm for language processing 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 employing Scala to write the whole attribute grammar using a wide range of modern attribute grammar concepts. The resulting system is lightweight and easy to understand, yet capable of competing in expressivity with systems such as JastAdd, a mature generator-based system which uses a similar evaluation method. A limited performance comparison with JastAdd showed that the Kiama approach is not impractical, but a comprehensive comparison is yet to be done.

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. Implicit parameters enable modularity via attribute forwarding.

Kiama is in active development. For example, we are adding collection attributes [38,39]. Scala's ability to extend traits that define values (as opposed to methods) will hopefully improve, so that we can 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.

Acknowledgements

Matt Roberts and the anonymous referees made many suggestions that improved the paper, for which we are grateful.

References

[1] P. Deransart, M. Jourdan, B. Lorho, Attribute Grammars: Definitions, Systems and Bibliography, in: Lecture Notes in Computer Science, vol. 323, Springer-Verlag, Berlin, Germany, 1988.
[2] J. Paakki, Attribute grammar paradigms—a high-level methodology in language implementation, Computing Surveys 27 (2) (1995) 196–255.
[3] M. Jourdan, An optimal-time recursive evaluator for attribute grammars, in: Proceedings of the International Symposium on Programming, Springer, 1984, pp. 167–178.
[4] J. Saraiva, Purely functional implementation of attribute grammars, Ph.D. Thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.
[5] G. Hedin, E. Magnusson, JastAdd: an aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37–58.
[6] A. Baars, D. Swierstra, A. Löh, UU AG System User Manual, Information and Communication Sciences, Faculty of Science, Utrecht University, 2011.
[7] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Science of Computer Programming 75 (1+2) (2010) 39–54.
[8] O. de Moor, K. Backhouse, S. Swierstra, First-class attribute grammars, Informatica 24 (3) (2000) 329–341.
[9] P. Hudak, Modular domain specific languages and tools, in: Proceedings of the 5th International Conference on Software Reuse, IEEE Computer Society Press, 1998, pp. 134–142.
[10] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, Computing Surveys 37 (4) (2005) 316–344.
[11] A. Sloane, Lightweight language processing in Kiama, in: Generative and Transformational Techniques in Software Engineering III, in: Lecture Notes in Computer Science, vol. 6491, Springer, 2011, pp. 408–425.
[12] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima Press, 2008.
[13] Scalatest, http://www.scalatest.org/.
[14] Specs, http://specs.googlecode.com/.
[15] D. Spiewak, T. Zhao, ScalaQL: language-integrated database queries for scala, in: Software Language Engineering, in: Lecture Notes in Computer Science, vol. 5969, Springer, 2010, pp. 154–163.
[16] G. Dubochet, On embedding domain-specific languages with user-friendly syntax, in: 1st Workshop on Domain-Specific Program Development, 2006, pp. 19–22.
[17] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1989, pp. 131–145.
[18] G. Hedin, Reference attributed grammars, Informatica 24 (3) (2000) 301–317.
[19] T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: International Summer School in Generative and Transformational Techniques in Software Engineering, in: Lecture Notes in Computer Science, vol. 4143, Springer, 2006, pp. 422–436.
[20] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proceedings of the International Conference on Compiler Construction, in: LNCS, vol. 2304, Springer, 2002, pp. 128–142.
[21] E. Magnusson, G. Hedin, Circular reference attributed grammars—their evaluation and applications, Electronic Notes in Theoretical Computer Science 82 (3) (2003) 532–554.
[22] L.C.L. Kats, A.M. Sloane, E. Visser, Decorated attribute grammars. Attribute evaluation meets strategic programming, in: Proceedings of the International Conference on Compiler Construction, in: Lecture Notes in Computer Science, vol. 5501, Springer, 2009, pp. 142–157.
[23] R. Bird, Using circular programs to eliminate multiple traversals of data, Acta Informatica 21 (3) (1984) 239–250.
[24] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming 72 (1–2) (2008) 52–70.
[25] E. Nilsson-Nyman, G. Hedin, E. Magnusson, T. Ekman, Declarative intraprocedural flow analysis of Java source code, in: Proceedings of the Workshop on Language Descriptions, Tool and Applications, 2008.
[26] U. Kastens, A.M. Sloane, W.M. Waite, Generating Software from Specifications, Jones and Bartlett, Sudbury, MA, 2007.
[27] U. Kastens, W.M. Waite, Modularity and reusability in attribute grammars, Acta Informatica 31 (1994) 601–627.
[28] M. Odersky, M. Zenger, Scalable component abstractions, in: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications, 2005, pp. 41–57.
[29] R. Mariani, Garbage collector basics and performance hints, MSDN Library, http://msdn.microsoft.com/en-us/library/ms973837.aspx (April 2003).
[30] D. Wampler, A. Payne, Programming Scala, O'Reilly, 2008.
[31] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, ACM, New York, NY, USA, 2007, pp. 1–18.
[32] PicoJava checker, http://jastadd.cs.lth.se/examples/PicoJava/.
[33] T. Johnsson, Attribute grammars as a functional programming paradigm, in: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag, London, UK, 1987, pp. 154–173.
[34] A. Augusteijn, Functional programming, program transformations and compiler construction, Ph.D. Thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1993.
[35] O. de Moor, S. Peyton-Jones, E. Van Wyk, Aspect-oriented compilers, in: Proceedings of International Symposium on Generative and Component-based Software Engineering, in: Lecture Notes in Computer Science, vol. 1799, Springer, 1999, pp. 121–133.

[36] E. Visser, Z. el Avidine Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: International Conference on Functional Programming, ICFP 1998, ACM, 1998, pp. 13–26.
[37] M. Viera, S.D. Swierstra, W. Swierstra, Attribute grammars fly first-class: how to do aspect oriented programming in Haskell, in: Proceedings of the International Conference on Functional Programming, ACM, 2009, pp. 245–256.
[38] J.T. Boyland, Remote attribute grammars, Journal of the ACM 52 (4) (2005) 627–687.
[39] E. Magnusson, T. Ekman, 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, IEEE Press, 2007, pp. 69–80.