Respect Your Parents: How Attribution and Rewriting Can Get Along

Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey

Department of Computing, Macquarie University

Abstract. Attribute grammars describe how to decorate static trees. Rewriting systems describe how to transform trees into new trees. Attribution is undermined by rewriting because a node may appear in both the source and product of a transformation. If an attribute of that node depends on the node's context, then a previously computed value may not be valid. We explore this problem and formalise it as a question of ancestry: the context of a node is given by the tree's parent relationships and we must use the appropriate parents to calculate attributes that depend on the context. We show how respecting parents naturally leads to a view of context-dependent attributes as tree-indexed attribute families. Viewed in this way, attribution co-exists easily with rewriting transformations. We demonstrate the practicality of our approach by describing our implementation in the Kiama language processing library.

1 Introduction

Tree attribution and tree rewriting are two fundamental paradigms of software language engineering. On the one hand, attribution focuses on calculating properties of a given program represented as a syntax tree. Attribute grammars are a standard way to declaratively specify the way in which trees should be attributed. On the other hand, rewriting concentrates on transforming a program represented by a syntax tree into a program or other artefact represented by another tree.

Attribution and rewriting are more usefully deployed together to solve a language engineering task. For example, in a compiler we might obtain an initial tree from a syntax analyser and perform some attribution on it to check some basic static properties. We might transform the initial tree into another one, perhaps to desugar complex constructs into simpler ones. Following this transformation, we might calculate attributes of the desugared tree to determine information that is needed for a further transformation process that produces the output we desire such as compiled code.

The ease with which such a process can be described is deceptive, since the detail of combining attribution and rewriting contains a subtle trap. It is common for rewriting implementations to use immutable data representations and deploy structure sharing since duplication of nodes can lead to significant extra memory use [1]. For example, a desugaring transformation might retain some parts of the input tree if those parts do not contain any complex constructs.

If nodes are shared between trees, how do we think of their attributes? These nodes are unchanged by the rewriting process and are shared by the before and after trees for efficiency reasons. A shared node may have different ancestors in the two trees and attributes that depend on these ancestors may well have different values depending on which set of ancestors we use. For example, the type of an expression may be different after rewriting if the type of a variable it uses has been changed. The type of the variable is given by the context of the expression in a particular tree, not as a property of the expression node itself. The core problem is to identify which attributes are valid after rewriting and which are not so that we do not need to recompute valid ones and we can avoid using invalid ones.

In this paper we describe how we addressed this problem in the context of our Kiama Scala-based language processing library [2], its implementation of attribute grammars [3] and its use of strategic term rewriting for transformation. Since it is based on Scala, Kiama uses object representations for trees and reference equality to implement node identity. Our aim was to develop a disciplined way to structure Kiama-based attribution under these conditions so that we could perform arbitrary rewriting of trees without invalidating previously calculated attribute values or obscuring the identity of the trees in which attribute values were valid.

The essence of our approach is that attribution should be performed with respect to the whole tree, not just with respect to a node in the tree (Section 2). Context-dependent attributes rely on the parent relationships to explore the context and its properties. It is the tree that defines the parent relationships between nodes, not the nodes themselves. If we base context-dependent attribution solely on the identity of a node then we are asking for trouble since that node may have more than one context if it is shared.

Our technical solution is to define context-dependent attributes as tree-indexed attribute families, not as single attributes (Section 3). To use one of these attribute families we must supply a tree to get an attribute that is valid for computations within that tree. At that point, regular attribute evaluation takes over without change. With attribute families, it becomes impossible to use a context-dependent attribute without thinking about the tree in which it is computed. Moreover, this tree-based focus is completely independent of the mechanism by which the tree was obtained. We can use any rewriting process we like without affecting the attribution.

We have implemented this approach in Kiama (Section 4). We previously relied on a mutable parent field in each tree node which was updated after a rewriting step. Because the parent had potentially changed it was necessary to erase all attribute values in case they were now invalid in the rewritten tree. Our changes mean that the mutable parent field has been removed, but no other changes were necessary to the core of the attribution and rewriting components. All of the existing Kiama test specifications were easily moved to the new scheme. We can now interleave attribution and rewriting of trees that share nodes without any danger.

We compare our approach to those of related attribute grammar-based systems that feature some element of rewriting or tree transformation (Section 5). We believe our approach is the first time that a general scheme for attribution has been given that can interoperate safely with arbitrary rewriting without the implementations of attribution or rewriting being dependent on each other.

2 Background

We begin by discussing examples of attribution and rewriting of simple tree structures to illustrate where problems can occur. This discussion motivates our solution which we describe informally here and more formally in the next section.

Our examples are based on simple arithmetic expressions that conform to the following context-free syntax rules:

$$\begin{aligned} \mathbf{Top} &: \mathbf{Root} ::= \mathbf{Node} \\ \mathbf{Num} &: \mathbf{Node} ::= \mathbf{Int} \\ \mathbf{Plus} &: \mathbf{Node} ::= \mathbf{Node}\ \mathbf{Node} \end{aligned}$$

Thus, the tree

$$\mathbf{Top}(\mathbf{Plus}(\mathbf{Plus}(\mathbf{Num}(1), \mathbf{Num}(2)), \mathbf{Plus}(\mathbf{Num}(3), \mathbf{Num}(4))))$$

represents the expression $(1 + 2) + (3 + 4)$ and is depicted in Figure 1.

2.1 Attribution

Suppose that we are interested in the *height* of nodes as measured by their maximum distance from a leaf. The following attribute grammar equations suffice to

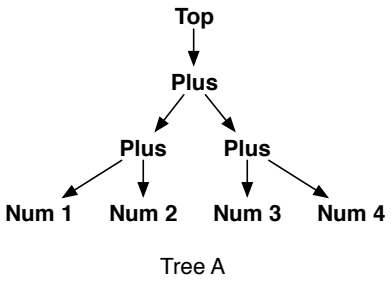

Fig. 1. Tree A represents the arithmetic expression $(1 + 2) + (3 + 4)$.

specify this attribute.[1]

$$\begin{aligned}&\mathbf{Num} : \mathbf{Node} ::= \mathbf{Int}\\&\quad \mathbf{Node.height} = 0\end{aligned}$$

$$\begin{aligned}&\mathbf{Plus} : \mathbf{Node}_1 ::= \mathbf{Node}_2\ \mathbf{Node}_3\\&\quad \mathbf{Node}_1\mathbf{.height} = 1 + \mathbf{max}(\mathbf{Node}_2\mathbf{.height}, \mathbf{Node}_3\mathbf{.height})\end{aligned}$$

The height of a **Num** is always zero since it is a leaf. The height of a **Plus** node is one more than the maximum of the heights of its children. Subscripts are used in the equations to distinguish between multiple occurrences of the **Node** symbol in **Plus** rule. Applying these equations in Tree A tells us that the height of node 1 is two and the height of node 4 is zero.

In effect, the height attribute equations define a pattern of computation that proceeds upward in the tree from the leaves to the node of interest. Traditionally, this kind of attribute is called a *synthesized attribute*.

In contrast, some attributes naturally depend on the context of the node at which they are computed and the information flows downward in the tree. For example, consider calculating the *depth* of a node in a tree which is its distance from the root. In traditional terminology, depth is an *inherited attribute* whose definition is given by equations that are associated with every rule that specifies context for the **Node** symbol.

$$\begin{aligned}&\mathbf{Top} : \mathbf{Root} ::= \mathbf{Node}\\&\quad \mathbf{Node.depth} = 0\end{aligned}$$

$$\begin{aligned}&\mathbf{Plus} : \mathbf{Node}_1 ::= \mathbf{Node}_2\ \mathbf{Node}_3\\&\quad \mathbf{Node}_2\mathbf{.depth} = \mathbf{Node}_1\mathbf{.depth} + 1\\&\quad \mathbf{Node}_3\mathbf{.depth} = \mathbf{Node}_1\mathbf{.depth} + 1\end{aligned}$$

These equations explain why the **Top** production is needed. Without it, there would be no context for the topmost **Node**. The **Top** context defines a depth of zero for its constituent **Node**, whereas a **Plus** context increments the depth by one. Applying these equations in Tree A tells us that the depth of node 1 is zero and the depth of node 4 is two.

The height and depth attributes are simple but attributes like them are the basis of any attribute grammar. Information is propagated up or down the tree from the place where it is available to where it is needed. A typical synthesized attribute is the value of a constant expression in a programming language. Name analysis can be performed using attributes that propagate information about declarations up to nodes that define scopes and then down to nodes that represent uses.

[1] Throughout the paper we use a generic attribute grammar notation that can easily be translated into the notations of particular tools. Context-free grammar rules are augmented by equations that specify how to calculate attributes of tree nodes using constants, pre-defined operations and the values of attributes at other nodes.

Modern attribute grammar systems build more advanced concepts on top of synthesized and inherited attributes, such as short-hand notations to make it easier to transport information up and down the tree, and attributes defined by fixed point computation. Extensions such as reference attributes and circular attributes are also supported by Kiama and similar systems. In this paper we focus on simple attributes in our examples, but the technique extends to more complex ones, since it does not affect the attribute evaluation mechanisms.

2.2 Rewriting

With the height and depth attributes in place, we now consider a rewriting transformation. The left-hand side of Figure 2 shows Tree A, repeated from Figure 1 with node numbers added for identification. The right-hand side of Figure 2 shows Tree B, a possible result of rewriting Tree A. In Tree B a new Plus node with a zero left child has been added at the top, and the right-most leaf has been incremented. These changes are typical of the effects of rewriting: embedding an existing tree in a new context, and changing a deeply-nested subtree.

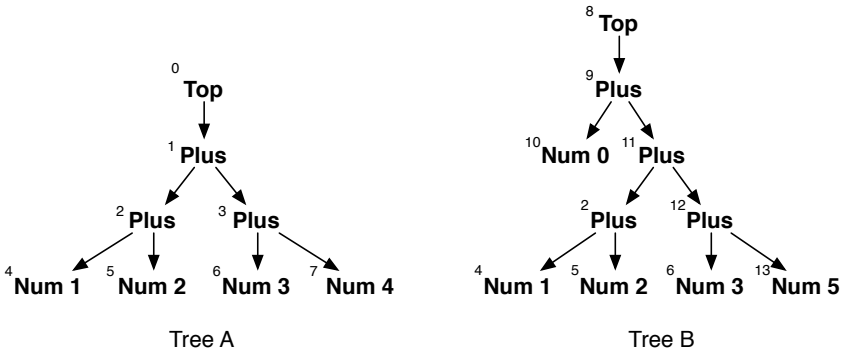

Fig. 2. Trees A and B represent arithmetic expressions $(1+2)+(3+4)$ and $0+((1+2)+(3+5))$, respectively. The superscripts number the individual nodes for identification in the text.

If we assume that sharing is allowed, that Tree B was produced from Tree A, and that trees are immutable, Tree B can share nodes 2, 4, 5 and 6 with Tree A. Nodes 9 and 10 in Tree B have no counterpart in Tree A. Node 13 results from rewriting node 7. Whenever a node is rewritten, all its ancestor nodes must also be replaced because they have at least one new child, even though they have not been explicitly transformed. This is the reason for replacing nodes 1, 3 and 7 by nodes 8, 11 and 12.

2.3 The problem and solutions

The central problem that we aim to address is how to compute the attributes of nodes that are shared between two trees. In some cases, the way that the attribute is computed means that its value cannot be affected by the sharing. Specifically, computation at a shared node n of an attribute that is only dependent on information from the sub-tree rooted at n cannot be influenced by rewriting since that sub-tree has not changed. For example, the height attribute is such an attribute and it is easy to see that the heights of nodes 2, 4 and 5 are the same in both Tree A and Tree B. In other cases, the nature of the attribute is to depend on the context of the node at which it is evaluated. A shared node might have different contexts in the two trees and hence different values for the attribute in those trees. For example, the depth of node 2 in Tree A is one, but in Tree B it is two since an extra node has been added above it.

The simplest approach to combining attribution and rewriting in the presence of sharing is to just calculate all of the attributes during a traversal of the tree from the root. It does not matter if a node is shared if we reach it via a path from the root in the relevant tree since that path gives us the context in that tree. Modern attribute grammar systems tend to prefer a more dynamic approach where attribute occurrences are only evaluated when needed [4, 5, 3]. Conceptually we ask a node for the value of one of its attributes which might trigger evaluation of attributes of other nodes. A primary motivation for this form of evaluation is interactive applications where we want to respond as quickly as possible with just what is needed. For example, in a development environment if we want to display a tool-tip for the code under the mouse pointer, we don't want to wait for a traversal to calculate every attribute if we can get the appropriate tool-tip with a much smaller set.

Assuming that we don't want to re-evaluate all of the attributes, we need a way to evaluate an attribute in a rewritten tree without having the context be an implicit piece of information in the evaluator. Our key observation is that attributes like depth depend on the parent relationships of the tree in which it is used. We must *respect the parents* in order to get a sensible result. Accordingly, our general solution is to regard context-dependent attributes as being parameterised by the tree in which they are being evaluated. In other words, instead of being attributes, they are tree-indexed attribute families. We must first supply the relevant tree and then we get an attribute that can be used safely for computations in that tree.

This approach has three main advantages. First, attribute families that are tree-indexed cannot be used without explicitly supplying the appropriate tree. This requirement removes the possibility of confusion that is present if the context is a property of tree nodes independent of the trees in which they occur. Second, because the tree has a separate identity to the nodes within it, a node can participate in more than one tree without problem. Significantly, attribution and rewriting do not need to be aware of each other, yet can operate together. Third, not having to erase attributes after rewriting should lead to efficiency gains since attributes that are still valid do not have to be re-calculated.

2.4 Kiama

Our main practical motivation for this work was to improve the implementation of attribute grammars in the upcoming 2.0 release of our Kiama language processing library [2, 3]. Kiama combines attribute grammars with strategic term rewriting in the style of Stratego [6]. We aimed to make context-dependent attributes safer when used in concert with rewriting based on generic tree traversals.

Kiama departs somewhat from the traditional view of attribute grammars used above to define the height and depth attributes. Instead of being defined by associating equations with grammar productions, Kiama attributes are defined by pattern matching against the tree structure at the node of interest. When a pattern matches a node, then the corresponding expression is used to calculate the value of the attribute at the node. In effect, a pattern and corresponding expression together define an equation for the attribute. Section 3 formalises the relevant aspects of this way of writing attribute grammars and gives examples.

Kiama's focus on pattern matching to decide which equation to apply means there is no clear distinction between synthesized and inherited attributes. In fact, one equation for an attribute can use the context to define the attribute value (inherited aspect) while another equation for the same attribute can use just child nodes (synthesized aspect). In a traditional setting the synthesized and inherited aspects would need to be split into separate attributes since one would need to be defined in the context of the relevant node. A consequence of Kiama's approach is that it is not necessary to introduce extra context productions in order to specify inherited attributes of the root of the tree. For example, we can define the expression example from the previous sections without needing the extra **Top** production to add context at the root of the expression tree. Section 3 shows how this can be done.

The attribution libraries in Kiama 1.x assume that the tree nodes contain a mutable parent field [3]. After a rewriting step, the parent fields must be updated to reflect the participation of shared nodes in the new tree, since those fields represent the parent relationships of the old tree. Updates to the parent fields potentially invalidate previously computed attribute values. To be safe, we must currently erase all of those values. There is no static check that this operation is performed and subtle bugs can be created if it is omitted. Even if the erasure is performed, we waste effort if erased values would have still been valid in the new tree. Moreover, after rewriting we cannot compute context-dependent attributes in the old tree at all since its parent information in shared nodes has been overwritten.

The approach developed in this paper removes these drawbacks. Kiama no longer assumes the existence of mutable parent fields. It is not possible to access a context-dependent attribute without specifying which tree is relevant. Computed attribute values remain valid after rewriting since the parent relationships from the old tree are still valid. We can calculate attributes on the old tree after rewriting just by using the old tree's parents. Attribution and rewriting don't have to know details of each other's implementation. In summary, our new ap-

proach means that attribution and rewriting can be freely mixed and that the possibility of bugs due to subtleties of their interaction is greatly diminished.

3 How to respect your parents

In this section we make these ideas concrete by formalising Kiama-style attribute grammars that respect their parents. We are only concerned with the dynamic evaluation behaviour of our attribute grammars. Therefore, we simplify the presentation by assuming that they do not contain any static errors that in Kiama would be ruled out by the Scala compiler. For example, we assume that the patterns in attribute definitions are non-linear and that the right-hand side of a case does not refer to unbound variables. We assume that constructors are always applied to the correct number of arguments in tree construction and in patterns. We do not consider aspects such as which attributes are defined for which node types since these aspects are orthogonal to our main topic.

3.1 A core attribute grammar language

Figure 3 summarises the abstract syntax of programs in a core attribute grammar language that is consistent with Kiama. The core language omits more complex Kiama attributes such as reference, higher-order and circular attributes. Each of these kinds of attribute can be incorporated into our scheme with no extra mechanisms and we have done so in our implementation.

A program consists of one or more definitions of trees and attributes, followed by one or more expressions that calculate values using those definitions. Definitions can bind variables to tree values. We assume that x ranges over the names of global variables and over the names of variables bound by pattern matching. Trees are defined over constructors C_n of arity $n >= 0$ that we assume are pre-defined and fixed. A tree is created by an application $C_n(t_1, \ldots, t_n)$ of a constructor C_n to sub-trees $t_1, \ldots, t_n$.

Program	$p ::= d^+ e^+$	
Definitions	$d ::= x = t$	tree-valued binding
	$\mid\ a = c$	attribute definition by cases
Trees	$t ::= C_n(t_1, \ldots, t_n)$	construction
Cases	$c ::= \textbf{case}\ p \to e$	match and evaluate
	$\mid\ c\ c$	sequence
Patterns	$p ::= x$	variable pattern
	$\mid\ C_n(p_1, \ldots, p_n)$	constructor pattern
Expressions	$e ::= f_n(e_1, \ldots, e_n)$	function call
	$\mid\ x.a$	evaluate attribute

Fig. 3. Abstract syntax of the core attribute grammar language.

Definitions can also bind attribute names to equations given by one or more pattern matching cases. The meta-variable a ranges over the names of attributes. Each case of an attribute definition specifies a match of a pattern against the node at which the attribute is being evaluated. If the pattern matches, the corresponding expression is evaluated to determine the value of the attribute at that node. Cases are applied in program order. Patterns are either variable patterns x which match any tree, or constructor patterns $C_n(p_1, \ldots, p_n)$ which match only trees whose root is formed by the constructor C_n and whose children match the patterns $p_1, \ldots, p_n$.

We assume f_n ranges over the names of globally available functions with arity $n >= 0$. An expression $f_n(e_1, \ldots, e_n)$ applies function f_n to the expressions $e_1, \ldots, e_n$. To simplify the presentation we assume that we can use standard mathematical functions using infix notation. We also assume the existence of a function for conditional expression evaluation written $e \ ? \ e \ : \ e$ which only evaluates one of its second and third arguments.

An expression $x.a$ evaluates the attribute a at node x. Evaluation of this form of expression involves applying the definition of attribute a to the node bound to variable x.

We can write the height attribute from Section 2 using the core language as follows.

$$\begin{array}{l}\textbf{height} = \\ \quad \textbf{case } \textbf{Num}(i) \rightarrow 0 \\ \quad \textbf{case } \textbf{Plus}(l, r) \rightarrow 1 + \textbf{max}(l.\textbf{height}, r.\textbf{height})\end{array}$$

3.2 Parents as node properties

One way to incorporate access to parents in this Kiama view of attribute grammars is to regard them as being properties of the tree nodes. Formally, we can assume that there is a global function **parent**(x) that returns the parent of a node x. We assume an auxiliary function **isRoot**(x) that returns true if and only if **parent** is not defined at x.

We can write the **depth** attribute from Section 2 as follows using the **parent** and **isRoot** functions.

$$\begin{array}{l}\textbf{depth} = \\ \quad \textbf{case } n \rightarrow \textbf{isRoot}(n) \ ? \ 0 \ : \ \textbf{parent}(n).\textbf{depth} + 1\end{array}$$

As discussed in Section 2, the problem with this approach is that a given node x may participate in more than one tree. Which parent do we get when we call **parent**(x)? Which root returns true from **isRoot**? If the parent property cannot change, then presumably we get the parent of the first tree in which x participates. We will not be able to correctly compute attributes for later trees. If the parent property is mutable, then we have to be careful to compute only attributes on the old tree before the property changes and only attributes of the new tree after the change. This dependence on mutability makes attribute computations fragile.

3.3 Parents as tree properties

Our solution is to focus on the parent relationships of a tree, rather than on the parents of nodes. The parent relationships of a tree can be calculated by traversing from the root, if they are not otherwise available. Thus, **parent** is now a function from the relevant tree to the parent partial function for that tree. We now write $\mathbf{parent}(x_1)(x_2)$ to get the parent of node x_2 in the tree that is rooted at the node bound to x_1. $\mathbf{isRoot}(x_2)$ becomes $\mathbf{isRoot}(x_1)(x_2)$ where x_1 is the root of the relevant tree; this operation can be implemented by a simple reference equality test.

In this new scheme, the definition of **depth** must be modified to have access to the current tree that is being attributed since it needs to use that tree's parent relationships. A simple way to think of this modification is that the depth attribute becomes an attribute family indexed by the tree. In other words, we don't just have one depth attribute, we have one for each possible tree.

We formalise attribute families by extending the core language to include a new definition form.

Definitions	$d ::= \ldots$		previous forms
	$\mid$	$a(x) = c$	attribute family

In the attribute family form, the variable x refers to the tree rooted at the node bound to variable x. We also need a new expression form to pass the tree rooted at x_1 to a family a to get an instance that can be evaluated at the node bound to x_2.

Expressions	$e ::= \ldots$		previous forms
	$\mid$	$x_2.a(x_1)$	instantiate attribute family and evaluate

With these extensions, the definition of the depth attribute becomes

$$\begin{array}{l}\mathbf{depth}(x_1) = \\ \quad \mathbf{case}\ x_2 \rightarrow \mathbf{isRoot}(x_1)(x_2)\ ?\ 0\ :\ \mathbf{parent}(x_1)(x_2).\mathbf{depth}(x_1) + 1\end{array}$$

Thus, the attribute is now insulated against tree changes since it is statically impossible to use **depth** without specifying the relevant tree.

3.4 Discussion

The key benefit of the attribute family approach is that by construction we rule out accessing the parent of the wrong tree, rather than allowing access to the parents at any time and relying on discipline to access them only at an appropriate time.

If we are defining an attribute that does not use the context, it can be defined by a regular definition since it does not need the tree. If we need the context, then we must use an attribute family and give a name to the context in the family definition. The dependence on the tree is now explicit and the user of an attribute family is required to provide the appropriate tree.

What about an attribute a_1 that does not require the context directly but whose definition uses an attribute a_2 that does require the context? We can choose from a couple of options depending on the situation. The first option should be used when it is meaningful for a_1 to be defined with respect to a specific context, not for all trees. We would define a_1 using a normal definition and pass that specific context when invoking a_2. For example, this case occurs when code has been transformed in a way that changes types but error messages should refer to user-specified types as defined by the original tree. The second option should be used when a_1 must be defined for all contexts and, if being applied in tree t, calls to a_2 should use t too. In this case we would define both a_1 and a_2 as families.

Another issue in the definition of an attribute like **depth** above is what happens if the node x_2 is not actually in the tree rooted at x_1? In this case x_2 is not the root and the **parent**(x_1) relation is not defined at x_2. Since our setting is a pure embedding of attribute grammars in another language, there is no easy way to statically prevent this situation. Nodes can be created at any time and the host language provides no connection between a node and the tree(s) that it is in. We currently ensure that **parent** and similar functions cause a run-time error if passed a node that is not in the tree which they are using. We are investigating ways to use Scala's type system to check for this situation statically. A similar approach based on a separately-defined attribute grammar language could build more safe-guards into the specification language.

4 Kiama Implementation

We now describe how we implemented the approach from the previous section. Kiama is a library for the Scala programming language [7] so we are able to use Scala's general-purpose facilities to implement attribute families. Kiama's existing attribute implementation was minimally affected by the changes. We just removed the implementation of the mutable parent field and information derived from it. Kiama's rewriting library was unaffected by the changes.

4.1 Relations

The base of our implementation is a new generic `Relation[T,U]` type defined over two types `T` and `U` (Figure 4). A relation is created from a sequence of tuples that define its graph. The operations are derived from the graph and are standard. For example, `compose` allows a relation to be composed with another that has a compatible type.

Because Kiama is based on Scala it is easy to provide relations with pattern matching support. Scala supports user-defined pattern matching via extractors [8]. We use extractors to allow any relation to be used in a pattern. For example, if R is a relation, then the pattern $R(p)$ will succeed if and only if R contains only a single tuple where the first component matches the node to which the pattern is applied and the second component matches p. The pattern

```
class Relation[T,U] (val graph : Seq[(T,U)]) {

    // Composition
    def compose[S] (st : Relation[S,T]) : Relation[S,U]

    // Domain
    def domain : Seq[T]
    def containsInDomain (t : T) : Boolean

    // Range
    def range : Seq[U]
    def containsInRange (u : U) : Boolean

    // Image and pre-image
    def image (t : T) : Seq[U]
    def preimage (u : U) : Seq[T]

    // Invert
    def invert : Relation[U,T]

    // Union
    def union (r : Relation[T,U]) : Relation[T,U]

}
```

Fig. 4. Part of Kiama's `Relation` interface. A relation is defined over the generic types `T` and `U`.

R.pair allows matching patterns against both the first and second components; *R.pair*(p_1, p_2) will succeed if and only if R contains only a single tuple where the first component matches pattern p_1 and the second component matches p_2. We show concrete examples of using this sort of pattern matching in Section 4.3.

4.2 Trees

The `Tree` class uses the general relation type to provide access to trees and their node relationships. Figure 5 shows representative parts of the `Tree` interface. A `Tree[T,U]` is created by providing the root value of some type `U`. The base type of all tree nodes is some other type `T` and we require that `U` is a sub-type of `T` (i.e., `U <: T`).

The base node type `T` is required to be a sub-type of Scala's `Product` type which enables us to determine the tree structure generically. Product values have generic access to their component fields. The `child` relation is computed by traversing throughout the tree from the root collecting pairs of nodes where one is a direct descendant of the other. We compute this value lazily since there is no need to perform that traversal if we don't use the child relation.

The `Tree` class also provides a suite of other relations which are derived from `child`. The `parent` relation is just the inverse of `child`. `siblings` is calculated

```
class Tree[T <: Product,U <: T] (val root : U) {
    // Base child relation
    lazy val child : Relation[T,T]

    // Derived relations
    lazy val parent : Relation[T,T]
    lazy val siblings : Relation[T,T]

    // Properties
    def index (t : T) : Int
    def isFirst (t : T) : Boolean
    def isLast (t : T) : Boolean
    def isRoot (t : T) : Boolean
}

object Tree {
    def isLeaf[T <: Product] (t : T) : Boolean
}
```

Fig. 5. Part of Kiama's `Tree` interface. Generic type `T` is the base type of tree nodes and type `U` is the type of the root node. The `Tree` object provides operations that do not depend on a specific tree.

by composing the `parent` relation with `child`. For example, if a is a child of b and b is a parent of c then a is a sibling of c. Other similar derived relations not shown in the figure give access to previous and next node, and so on. All of these relations are computed lazily since they might not be needed.

Some node properties are not dependent on the tree since they only depend on components of the node or its children. (Recall that nodes are immutable so these factors cannot change if a node is shared among trees.) For example, whether or not a node is a leaf cannot change if that node appears in more than one tree. Tree-independent operations such as `isLeaf` are static methods that accompany the `Tree` class.

4.3 Examples

In the Kiama setting, a tree-indexed attribute is just a class or a method that takes a `Tree`-value argument. For example, we can define the height and depth attributes from earlier sections as shown in Figure 6.

`height` is a regular attribute defined as it would be with Kiama 1.x. `attr` is the Kiama attribute creation method which takes a single argument that is a collection of cases to specify the attribute equations. `attr` implements attribute caching and dynamic circularity testing on top of the equation definitions. We made no changes to `attr` for this present work.

```
val height : Node => Int =
    attr {
        case Num (_)     => 0
        case Plus (l, r) => 1 + height (l).max (height (r))
    }

class DepthModule (tree : Tree[Node,Node]) {

    def depth : Node => Int =
        attr {
            case tree.parent (p) => depth (p) + 1
            case _               => 0
        }

}
```

Fig. 6. Kiama version of the height and depth attributes.

In contrast to `height`, `depth` requires access to the context. In Figure 6, `depth` is defined in a class whose constructor takes the tree as an argument.[2] In effect, the class defines a reusable module of attribution. A client of this module would have to first instantiate the class with the desired tree. The definition of `depth` uses the tree to access the parent relationship. In the first equation the pattern `tree.parent (p)` will succeed only if the matched node has a single parent in that tree and it will bind that parent node to the variable `p`. The variable is used in the right-hand side of the equation `depth (p) + 1` to recursively get the depth of the parent. The second equation will only be reached if the node has no parent, which means it must the root of the tree.

Instead of defining a module of related attributes using a class, we could define a family for a single attribute by using a method that takes the tree as an argument. In our experience this approach is less useful than using a class, since it is common for many related attributes to need access to the tree. It is easier to group these attributes in a module and then pass the tree once when the module instance is created than it is to pass the tree explicitly to many separate attribute definitions.

Cooperation between different attribute families is achieved in different ways that depend on how the families are defined. If they are defined in the same module, then the context is implicitly available to both families, so it need not be passed. If the families are defined in different modules, then a calling attribute will need to be given a reference to the module instance that defines the called attribute. Similarly, a family defined by a method can call a family defined in a module if it has a reference to the relevant module instance. Finally, if two families are defined by parameterised methods, then the context will need to be

[2] The constructor arguments of a Scala class are given in the class heading and the body of the class definition is the constructor implementation. Constructor arguments are in scope throughout the class definition.

passed explicitly between them. Which of these situations applies will depend on the overall structure of an application, so it is hard to be definitive about the implication of use families. To give some expectations, we report in the next section on our experiences of converting Kiama's test suite to use the new approach.

4.4 Experience

We have converted our extensive Kiama test suite across to the new style of context-dependent attributes. The suite includes implementations of various languages including lambda calculus, Prolog and various cut-down versions of Java.

In all cases we have defined attribution modules that collect many related attributes, following the module pattern of Figure 6. Most of the code has not increased in size at all since we just converted singleton implementations of attribution modules into classes and now access the context in attribute equations via the tree's parent relation instead of via the tree node fields. A small code size increase is incurred where modules are instantiated since we must create an instance of the module instead of just accessing a singleton.

The biggest Kiama test is an Oberon-0 compiler that was previously built for the 2011 LDTA Tool Challenge. This compiler is built from more than twenty separate traits comprising around 2000 lines of Scala code. The traits are mixed together to form the artefacts required by the challenge. In the previous version, the attribution components relied on the `parent` field of nodes. In the new version the components are passed the relevant trees and, if they are transformation components, return new trees. For example, one component performs desugaring of FOR and CASE statements into WHILE and IF statements, respectively. The desugarer is given the input tree so it can use attributes that depend on it such as those supplied by name and type analysis. After the tree has been rewritten it is returned as a new tree that is then consumed safely by the next transformation or code generator. Previously, we needed to be careful to erase attributes of the old tree before the new tree was returned in case some of them were no longer valid.

Throughout our examples we now make extensive use of relational pattern matching where we need to check if a nearby node is there, optionally pattern match on it, and bind it. These patterns replace direct access to the parent via a tree node. The tree relations are the basis of properties such as `isRoot` which is true if the node is the root of the tree. The tree module also supplies operations that do not depend on the specific tree, but just on the node, such as `isLeaf`, `firstChild` and `lastChild`.

Nested patterns are particularly useful. For example, the pattern

```
u @ IdnUse (i1)
```

succeeds in the Oberon-0 compiler if matched against an identifier use node. It binds that node to the variable `u` and the identifier string to `i1`. The pattern

```
ProcDecl (IdnDef (i2), _, _, _)
```

matches procedure declaration nodes and binds the variable `i2` to the procedure identifier. We can nest these two patterns inside a parent pair pattern to help implement a check that the identifier used at the end of an Oberon-0 procedure declaration (`i1`) is the same as the one used in the procedure's heading (`i2`).[3]

```
case tree.parent.pair (u @ IdnUse (i1),
                       ProcDecl (IdnDef (i2), _, _, _)) =>
    message (u, s"end procedure name $i1 should be $i2",
             i1 != i2)
```

Kiama's message facility is used here to generate a message if `i1` and `i2` are not the same and place the message at the location of the identifier use (node `u`).

As another example, the following pattern was used in a dataflow example to see if the current node has both a next sibling (`n`) and a `Block` parent.

```
tree.parent.pair (tree.next (n), _ : Block)
```

In both of these examples, the explicit use of `tree` ensures that these context-dependent matches are performed with respect to the tree that was provided when these modules were created. We have thereby reduced the risk that we will accidentally check in the wrong tree.

A secondary benefit of having the tree available in an attribute definition is that we can directly refer to the root node. A direct reference can be used to short-cut the usual pattern of attribution where an attribute computed at the root node has to be transported one step at a time down to where it is needed.

5 Related Work

We focus our discussion on related work that substantially involves attribute grammars and that incorporates some aspect of tree transformation or rewriting in combination with attribution.

Incremental attribute evaluation. One approach to dealing with changes in attributed trees is to recompute attributes where necessary to take changes into account. A notable early example of incremental attribute evaluation is Reps' work to generate language-based editors that were specified using attribute grammars [9], but there are many later incremental approaches. Recent examples include work by Saraiva and Swierstra [10] and Bransen *et al.* [11]. Incremental evaluation requires some knowledge of attribute dependencies and the detail of tree changes in order to recalculate only when necessary. Bürger's RACR library for Scheme [12] is of particular interest since it incorporates arbitrary tree rewriting. RACR builds a dynamic attribute dependency graph during evaluation so it knows which attributes are influenced by rewrites. In contrast, our approach

[3] The relatively verbose form `tree.parent.pair` can be abbreviated by imports and aliases, but we choose to show the full form to keep the explanation simple.

might cause some unnecessary recalculation of context-dependent attributes in a rewritten tree, but we do not need to keep track of attribute dependencies or have any dependence between attribution and rewriting. It is future work to investigate how our approach can be adapted to share computed values between different instances of the same attribute family where it is safe to do so.

Object-based attribute grammar systems. Some attribute grammar systems generate evaluators for object-oriented languages and hence directly share some of the concerns of Kiama for mutability and shared tree nodes.

JastAdd has pioneered many recent extensions of the basic paradigm including reference and circular attributes [4, 13]. It generates evaluators in Java, including tree node classes that implement the attribute as methods. Each of these classes contains a mutable parent field, so tree nodes cannot participate in more than one tree. In addition to its main attribute grammar specification notations, JastAdd incorporates a form of rewrite rule [14]. It arranges to invoke these rules as part of the attribute evaluation process. Unfortunately, attributes of trees that are being rewritten may be recalculated as rewriting proceeds, only to be finalised when later rewriting cannot affect their values. We believe that this approach blurs the distinction between an attribute representing a property of a node and a mutable variable that may change as execution proceeds. In our scheme attributes will only ever have one value within a particular tree. Also, in JastAdd the rewriting approach is intricately embedded in the attribute evaluation process, whereas attribution and rewriting are independent in our approach.

Silver is another prominent Java-based attribute grammar system [5]. Silver distinguishes between trees that have no attribute values and ones that do (so-called "decorated trees"). Attributes are evaluated by passing them a reference to the tree context. Thus, in theory it is possible to decorate a node with respect to more than one tree by passing in a different context. However, as far as we can tell, Silver does not explicitly deal with node instances that are shared by two trees. Since attributes are computed lazily and their values stored, it would appear to be necessary to clear those values if we wanted to evaluate those attributes in another tree that shared some nodes, as in older versions of Kiama. Silver supports language extension via a form of higher-order attribute grammar called forwarding [15]. New tree fragments can be computed as attribute values that are associated with existing tree nodes and forward some attribution requests to those nodes. Forwarding, in essence, is a specialised form of tree transformation by augmentation and is supported directly by the Silver evaluation approach. In contrast, our approach can support arbitrary tree transformations that are independent of attribute evaluation.

Functional attribute grammar systems. There is a long tradition of attribute grammar systems based on or in functional programming languages [16]. By and large, these systems do not encounter the same issues with sharing since in pure value-based functional languages sharing is not observable. Hence, there

is no option to associate attribute information with a node instance since there is no way to tell that instance apart from another that has the same fields.

We briefly note two functional attribution approaches that have some characteristics in common with our approach. Zippers can be used in functional languages to keep track of the current location during a generic tree traversal [17]. Martins *et al.* use generic zippers to embed attribute grammars in a pure functional language [18]. During evaluation the zipper encodes the path taken from the root to the node of interest in a similar way to a traditional tree-walking attribute evaluator. It is non-trivial to start a zipper-based evaluator at a particular node as we do in our approach, since the context would have to be manually created. Accordingly zipper-based approaches assume a traversal from the root.

An alternative to zipper-based approaches for context tracking in functional languages was developed by Gaillourdet *et al.* [19]. A cyclic position structure is created to mirror the structure of the tree upon which attribution is to be performed. Nodes in the position structure have parent links to give access to the context of a node. This approach is similar to Kiama's previous approach in that it equips each node with a component that gives access to its parent in a particular tree structure. The functional setting means that quite a bit of work has to be done to define the form of a position structure based on the tree syntax and construct one for a particular tree. In contrast, our reference equality-based setting allows us to use the relationships between the nodes themselves and a separate mirroring structure is not needed.

Rewriting-based attribution systems. Kats *et al.* incorporated attribute grammar features into ASTER which is an extension of the Stratego strategic programming language [6, 20]. This combination of attribution on a rewriting base contrasts with Kiama's approach where attribution and rewriting can cooperate but are implemented separately. ASTER uses the generic traversal operators of Stratego to implement *decorators* that abstract patterns of attribute computation away from a specific tree structure. ASTER's focus on traversal from a node of interest is similar to Kiama's focus on the relationships between nodes. Reflection on the tree structure is used in ASTER to obtain access to a node's context via its parent. This reliance on a single parent reference means that ASTER cannot express attribution of shared nodes.

Relational representations of programs. Finally, we note that the use of relations to represent relationships between program components is a well-used idea. A non-trivial early example is Linton's OMEGA system which uses a relational database to store program information [21]. Since the aim of this kind of work is different from ours we do not consider it further, except to mention the Rascal language which incorporates high-level support for relations to support meta-programming [22]. It is possible that Kiama's new support for relations can be generalised beyond trees to support this kind of processing.

6 Conclusion and Future Work

We described how attribution and rewriting of trees can get along in an object-based implementation with reference equality. The key is to design context-dependent attribution to be parameterised by the tree in which that attribution is to be performed, thereby defining attribute families. This approach solves the problem of deciding what attributes mean when nodes are shared between trees as a result of rewriting.

We have implemented the approach in the Kiama language processing library and its test suite. Context-dependent attribute definitions were adjusted to depend on the tree, but no changes were needed in the attribute evaluation implementation so the approach works for all existing kinds of attributes including reference and circular attributes. The definition and evaluation of attributes is completely independent of how rewriting is achieved and requires no knowledge of which rewrites have occurred.

The main area for future work is to further improve reuse of attribute values in rewritten trees. At the moment we reuse all attributes that do not depend on the context. However, some context-dependent attribute occurrences will be valid in a rewritten tree if they do not use the part of the context that has changed. We are developing techniques to take advantage of this situation without requiring detailed cooperation between attribution and rewriting. Part of this work will be a detailed profiling exercise to understand how the evaluation of attributes is affected by the shift to attribute families.

Acknowledgements

Development of the approach described in this paper benefited greatly from discussions with our colleague Dominic Verity. We also thank the anonymous reviewers for their helpful suggestions.

References

1. van den Brand, M.G.J., Klint, P.: ATerms for manipulation and exchange of structured data: It's all about sharing. Information and Software Technology **49**(1) (2007) 55–64
2. Sloane, A.M.: Lightweight language processing in Kiama. In: Generative and Transformational Techniques in Software Engineering III. Volume 6491 of Lecture Notes in Computer Science. Springer (2011) 408–425
3. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure embedding of attribute grammars. Science of Computer Programming **78** (2013) 1752–1769
4. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming **47**(1) (2003) 37–58
5. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming **75**(1+2) (January 2010) 39–54

6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16: components for transformation systems. In: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, ACM (2006) 95–99
7. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. 2nd edn. Artima Press (2010)
8. Emir, B., Odersky, M., Williams, J.: Matching objects with patterns. In: Proceedings of European Conference on Object-Oriented Programming. Volume 4609 of Lecture Notes in Computer Science. (2007)
9. Reps, T.W.: Generating Language-based Environments. Massachusetts Institute of Technology, Cambridge, MA, USA (1984)
10. Saraiva, J., Swierstra, D.S., Kuiper, M.F.: Functional incremental attribute evaluation. In: Proceedings of the 9th International Conference on Compiler Construction, Springer (2000) 279–294
11. Bransen, J., Dijkstra, A., Swierstra, S.D.: Lazy stateless incremental evaluation machinery for attribute grammars. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation, ACM (2014) 145–156
12. Bürger, C.: RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting. Dresden University of Technology, http://racr.googlecode.com. (2014)
13. Magnusson, E., Hedin, G.: Circular reference attributed grammars–their evaluation and applications. Science of Computer Programming **68**(1) (2007) 21–37
14. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: Proceedings of the European Conference on Object-Oriented Programming. Volume 3086. (2004) 147–171
15. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proceedings of the International Conference on Compiler Construction. Volume 2304 of Lecture Notes in Computer Science., Springer (2002) 128–142
16. Johnsson, T.: Attribute grammars as a functional programming paradigm. Lecture Notes in Computer Science **274** (1987) 154–173
17. Adams, M.D.: Scrap your zippers: a generic zipper for heterogeneous types. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming, ACM (2010) 13–24
18. Martins, P., Fernandes, J., Saraiva, J.: Zipper-based attribute grammars and their extensions. In Bois, A., Trinder, P., eds.: Programming Languages. Volume 8129 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 135–149
19. Gaillourdet, J.M., Michel, P., Poetzsch-Heffter, A., Rauch, N.: A generic functional representation of sorted trees supporting attribution. In Voronkov, A., Weidenbach, C., eds.: Programming Logics. Volume 7797 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 72–89
20. Kats, L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: Proceedings of the International Conference on Compiler Construction. Number 5501 in Lecture Notes in Computer Science, Springer (2009) 142–157
21. Linton, M.A.: Implementing relational views of programs. In: Proceedings of the Symposium on Practical Software Development Environments, ACM (1984) 132–140
22. Klint, P., van der Storm, T., Vinju, J.: Easy meta-programming with Rascal. In: Generative and Transformational Techniques in Software Engineering III. Springer (2011) 222–289