Profile-based Abstraction and Analysis of Attribute Grammar Evaluation[*]

Anthony M. Sloane

Department of Computing, Macquarie University, Sydney, Australia
Anthony.Sloane@mq.edu.au

Abstract. Attribute grammars enable complex algorithms to be defined on tree and graph structures by declarative equations. An understanding of how the equations cooperate is necessary to gain a proper understanding of an algorithm defined by an attribute grammar. Existing attribute grammar tools and libraries provide little assistance with understanding the behaviour of an attribute evaluator. To do better, we need a way to summarise behaviour in terms of attributes, their values, their relationships, and the structures that are being attributed. A simple approach to program profiling is presented that models program execution as a hierarchy of domain-specific profile records. An abstract event for attribute evaluation is defined and evaluators are modified to collect event instances at run-time and assemble the model. A flexible report writer summarises the event instances along both intrinsic and derived dimensions, including ones defined by the developer. Selecting appropriate dimensions produces reports that expose complex properties of evaluator behaviour in a convenient way. The approach is illustrated and evaluated using the profiler we have built for the Kiama language processing library. We show that the method is both useful and practical.

1 Introduction

Attribute grammars promote a view of tree and graph decoration based on declarative equations defined on context-free grammar productions [18]. An attribute is defined by equations that specify its value at a node *N* as a function of constant values, the values of other attributes of *N*, and the values of attributes of nodes that are reachable from *N*. Provided that sufficient equations are defined to cover any context in which the node can occur, we obtain a declarative specification of an algorithm that can be used to compute the attribute of any such node. This approach to computation on structures has been shown to be extremely powerful. Recent applications that use attribute grammars heavily are XML integrity validation [2], protocol normalization [3], Java compilation [4], image processing [7], and genotype-phenotype mapping [11].

Any single attribute equation is usually fairly simple, but its effect is intricately intertwined with that of many other equations. The value of an attribute *A* is ultimately determined not just by *A*'s equations, but by the equations of any attribute on which *A*'s

[*] Matt Roberts, Nikki Lesley and Suzana Andova provided helpful feedback on this work. Much of the project was performed while the author was a guest of Mark van den Brand's group at TU Eindhoven. The author thanks the anonymous reviewers for their useful suggestions.

equations transitively depend. Thus, the computations that cooperate to compute a value of *A* are potentially dispersed throughout the attribute grammar. This dispersal means that it is non-trivial to determine what an attribute value is or even how an attribute is calculated just by looking at the equations.

Powerful extensions of the original attribute grammar formalism make this problem even worse. Equations in the original conception of attribute grammars can only refer to attributes of symbols that occur in the context-free grammar production on which the equation is defined. In higher-order and reference attribute grammars the value of an attribute can itself be a reference to a node upon which attributes can be evaluated [15, 27]. This extension is particularly useful when defining context-sensitive properties such as name binding or programmer-defined operator precedence. This power comes at a price, however, because it means that more parts of the grammar are in play when we are trying to understand a particular attribute and how it is computed.

The situation is complicated even more by the fact that many modern attribute grammar systems use a dynamically-scheduled evaluation strategy, which precludes accurate static analysis. Some classes of attribute grammar submit to static dependence analysis that enables evaluation strategies to be computed in advance of running the evaluator [12]. One can imagine tools based on this static analysis that would assist with understanding the evaluation process. More recently, however, attribute grammar tools and libraries have mostly used an approach where the evaluation strategy is determined at run-time [9, 10, 24, 26]. This approach admits more grammars than does a static approach, some algorithms are easier to express, and features such as higher-order and reference attributes are easier to support. However, a dynamically-scheduled approach also means that an accurate static analysis is not possible. Instead, dynamic analysis support is necessary since evaluation will be influenced by the attribute values, which will in turn be influenced by the input.

We therefore favour dynamic methods for understanding our attribute evaluators. In this paper we focus on a profile-based approach where data is collected at run-time and summarised to reveal relationships between different aspects of the execution. An option is to profile the code that implements the attribute evaluator. The resulting profiles are unsatisfactory, however, as they operate at a much lower level than the attribute grammar and require the developer to know a great deal about the implementation approach. Instead, we develop a method where both data collection and profile reports are in terms of an *abstract model of evaluation* that is based on arbitrary data *dimensions*. For example, the developer can ask for a profile that shows run-time and evaluation counts for each attribute that was evaluated during the run. Multi-dimensional reports show how dimensions are related to each other. For example, a report that summarises first along the attribute dimension and then along the subject (tree node) dimension gives insight into the places in the structure where the attributes were evaluated.

Specifically, the contributions of the paper are as follows.

1. The design of a general profiling framework based around generating a hierarchical model of program execution from domain-specific program events which have properties in arbitrary dimensions (Section 3.1).
2. The design of a profile reporting scheme that is independent of the event types produced by a program and of the dimensions possessed by the events (Section 3.2).

3. An implementation of the profiling framework and reporting scheme for attribute evaluators that use our Kiama language processing library [24] (Section 3.3).
4. An evaluation of the performance of the Kiama profiler that shows that it is sufficient for interactive use on large inputs (Section 3.4).
5. Examples of the approach and demonstration of its utility. We use the *PicoJava* Java subset and the *Oberon-0* variant of Oberon (Sections 2 and Section 4).

The paper concludes with a review of related work (Section 5) and an examination of directions for future work (Section 6).

Code and documentation for our implementation of the profiling framework can be found at `http://bitbucket.org/inkytonik/dsprofile`. Kiama can be obtained from `http://kiama.googlecode.com`.

2 Understanding Attribute Evaluation

To place our discussion on a concrete footing, we first illustrate the difficulties of attribute grammar understanding using the problem of name analysis for Java-like languages. In this section, we describe part of a standard attribute grammar solution for this problem and discuss how profiling can help us understand this attribute grammar.

2.1 PicoJava

The attribution we consider performs name analysis for the PicoJava language. This attribution was originally written as an illustration of reference attributes in the JastAdd system [8]. The Kiama distribution contains a fairly direct translation of the JastAdd attribute grammar. We present the attribute equations in a simplified system-neutral notation, however, since the problems of understanding the grammar and the profiling solution are independent of the details of the specific attribute grammar notation.

PicoJava contains declarations and uses of Java-like classes and fields, but omits most of the expression, statement and method complexity of Java. The left side of Figure 1 shows a PicoJava program consisting of a block with a declaration of class `A`. Class `A` contains two nested classes: `AA` and `AA`'s sub-class `BB`. Statements are limited to simple assignments between named objects which are either fields of the current object or qualified accesses to fields of other objects.

The problem that is solved by the attribute grammar is to check the use of all identifiers. For example, the uses of `x` in the `a.x` and `b.x` expressions are legal because of the declaration of the `x` field in `AA` and the inheritance relationship between `AA` and `BB`. However, the use of `y` in `b.y` is illegal since `BB` and `AA` declare no `y` field, even though there is a `y` field in the enclosing class `A`.

The attribute grammar operates on an abstract syntax tree representation of a program. The right side of Figure 1 shows the tree for the program on the left side. A program consists of a block containing a list of the top-level declarations and statements. Declarations are either of variables or of classes. A variable declaration (`VarDecl`) specifies the name of the variable and its type. Class declarations (`ClassDecl`) specify the class name and an optional name of the superclass. The superclass component is either `Some(c)`, if the superclass is `c`, or `None`, if there is no declared superclass. Uses of variable or class names are `Use` constructs. Classes contain a recursive block for their local declarations and statements. There is no limit to the nesting of class declarations.

2.2 Name analysis for PicoJava

The attribute grammar that implements name analysis for PicoJava defines one main attribute `decl` whose value is the declaration corresponding to a particular access of a name. `decl` is a reference attribute whose value is the actual `VarDecl` or `ClassDecl` node in the tree. `decl` is a *synthesized attribute*, meaning that it is defined in all productions that define the structure of accesses.

```
attribute decl : Access => Decl
Use: a:Access ::= u:Use          a.decl = u.lookup (u.name)
Dot: a:Access ::= Access u:Use    a.decl = u.decl
```

We first declare the type of the attribute. The `decl` attribute is defined on `Access` nodes and its type is `Decl`, the common superclass of `VarDecl` and `ClassDecl`. The arrow `=>` appeals to an intuition that attributes are functions from nodes to values.

PicoJava accesses come in two varieties: a direct use of a single name (`Use`) or a field reference with respect to a nested object access (a `Dot` containing an `Access` and a `Use`). In the attribute grammar the two varieties are represented by the two context-free grammar abstract syntax productions shown. Productions are written as the production name, a colon, then the grammar rule. The two parts of a grammar rule are separated by `::=`. The symbols in the grammar rule can be given names so that they can be referred

```
                          Program (
                           Block (List (
{                           ClassDecl ("A", None (),
 class A {                   Block (List (
  int y;                      VarDecl (Use ("int"), "y"),
  AA a;                       VarDecl (Use ("AA"), "a"),
  y = a.x;                    AssignStmt (
  class AA {                   Use ("y"),
   int x;                      Dot (Use ("a"), Use ("x"))),
  }                           ClassDecl ("AA", None (),
  class BB extends AA {        Block (List (
   BB b;                        VarDecl (Use ("int"), "x")))),
   b.y = b.x;                 ClassDecl ("BB", Some (Use ("AA")),
  }                            Block (List (
 }                              VarDecl (Use ("BB"), "b"),
}                               AssignStmt (
                                 Dot (Use ("b"), Use ("y")),
                                 Dot (Use ("b"), Use ("x")))))))))))
```

Fig. 1. A PicoJava program (left) and its abstract syntax tree (right).

to by the attribute equations. For example, in the first production the `Access` symbol is given the name `a`.

Since there are two productions that define the `Access` symbol, we need two definitions for the `decl` attribute. Each production has an equation that describes how to compute the attribute in an actual tree construct that was derived by that production. Each equation has the form *attribute* = *expression*. References to attributes of particular symbols are written using a "dot" notation on the left-hand side of an equation and within the right-hand side expression (e.g., `a.decl`). This notation is also used to access intrinsic properties of tree nodes that are already in the tree when attribution begins (e.g., the `u.name` field of a `Use`).

In order to describe how to compute an attribute value, an attribute equation can refer to any symbols of the associated production to obtain data values from attributes or intrinsic properties. Expressions can use any facility of the host environment to compute with these values. The overall effect of the `decl` equations is that the relevant declaration is determined by evaluating the `lookup` attribute at the rightmost name use in the access. For example, when evaluating `decl` for the expression `a.b.c`, which is represented by the tree `Dot(Use(a),Dot(Use(b),Use(c)))`, we will evaluate `decl` for `b.c`, `decl` for `c` and, finally, `lookup("c")` at the `Use(c)` node.

2.3 Name lookup

The `lookup` attribute implements a search to find the declaration that matches a particular name. The value of `n.lookup(s)` is a reference to the node that represents the declaration of `s` when viewed from the scope represented by the node `n`, or a reference to a special "unknown declaration" value if no such node can be found.

```
attribute lookup (String) : Any => Decl
```

We indicate that the attribute can be evaluated at any node using the `Any` common super-class of all tree node classes.

`lookup` is a *parameterised attribute*, since it depends on the name being sought. It is also an *inherited attribute* since its value will be determined by the context surrounding the node where it is evaluated, not by the inner structure of that node.

```
Use: a:Access ::= u:Use          u.lookup(name) = a.lookup(name)
Dot: Access ::= a:Access u:Use  u.lookup(name) =
                                    a.decl.type.remoteLookup(name)
```

The first of the two cases is when `lookup` is evaluated at a `Use` node. This corresponds to the use of an unqualified name, so we just continue the search at the parent node to search in the current scope. The other case is when the use is qualified, in which case it will occur as the child of a `Dot` node. We need to find the declaration of the object which is being accessed, get its type and perform a remote lookup there.

We omit the definitions of the `type` and `remoteLookup` attributes since those details are not necessary for our discussion. `type` returns a reference to a node representing the class type from a declaration. `remoteLookup` looks for a name in a class type from the perspective of a client of that type.

The remaining cases for `lookup` propagate requests coming from `Use` nodes to the appropriate parts of the tree and to trigger searches in blocks and super-classes.

```
...: ... ::= ... b:Block ...
    if (b.contains(name))
      b.lookup(name) = b.localLookup(name)

...: ... ::= ... c:ClassDecl ...
    if (c.superClass != null) && (c.superClass.contains(name))
      c.lookup(name) = c.superClass.remoteLookup(name)

...: p ::= ... n ...
    n.lookup(name) = p.lookup(name)
```

In these productions, we use an ellipsis `...` to indicate a part of a production whose structure is not important. Some of the equations are *conditional* in that they only apply if a Boolean condition is true. If the current node is a block and that block contains a definition of the name we are looking for, we perform a local lookup in that block to find the declaration. Otherwise, if we have reached a class declaration, that class has a super-class, and the super-class contains a definition of the name, then we perform a remote lookup in that class. The final case covers all other circumstances by just propagating the search to the parent node. We are guaranteed to eventually reach at least a block since all programs consist of one.

2.4 Understanding PicoJava name analysis

Name analysis attribution for PicoJava is a canonical example of reference attribute grammars [8]. The equations are not lengthy, but their operation is still quite hard to understand. Some of the difficulty is due to the inherent complexity of the problem being solved. The tasks of name and type analysis for a language like PicoJava are intertwined. For example, to lookup a name in the body of a class, we may need to search the superclass type, which involves performing name analysis on the name that appears in the superclass position of the class declaration, and so on, while avoiding problems such as cycles in the inheritance chain. (The latter check is hidden in the definition of the `superClass` attribute.)

Given this inherent complexity, it is somewhat surprising that the definitions are not longer than they are. The main reason for their brevity is the power of the dynamically-scheduled attribute grammar formalism to abstract away tree traversal details. When we are writing our equations, we can reason about how declarations, global, local and remote lookups, and types relate to each other, without having to work out a particular tree traversal that evaluates the attributes in the correct order. The attributes are evaluated when their values are first demanded and caching means that we don't need to worry about which particular use of an attribute asks for it first. In contrast, a solution based on tree traversals implemented by visitors, for example, would have to explicitly plan which attributes should be evaluated at which time. Developing such a plan is a non-trivial task for this problem.

```
    202 ms total time
     47 ms profiled time (23.5%)
    206 profile records

 By attribute:

  Total Total  Self  Self  Desc  Desc Count Count
     ms     %    ms     %    ms     %           %
     41  88.0    11  24.7    30  63.3    27  13.1  decl
     30  63.7    10  22.8    19  40.9    32  15.5  lookup
     15  32.9     9  19.0     6  13.9    19   9.2  localLookup
      5  10.6     0   1.8     4   8.8    18   8.7  unknownDecl
      4   8.9     4   8.9     0   0.0    33  16.0  declarationOf
      3   7.9     2   4.8     1   3.1     7   3.4  remoteLookup
      3   6.5     2   4.9     0   1.6     2   1.0  isSubtypeOf
```

```
 By type for lookup:

  Total Total  Self  Self  Desc  Desc Count Count
     ms     %    ms     %    ms     %           %
     33  60.8     7  13.2    26  47.7    12   5.8  Use
     17  31.8     1   3.2    15  28.6     4   1.9  VarDecl
      4   8.3     2   4.2     2   4.1     5   2.4  Block
      2   3.7     0   1.1     1   2.6     4   1.9  ClassDecl
      1   2.2     0   0.7     0   1.5     3   1.5  Dot
      1   2.1     0   0.8     0   1.3     4   1.9  AssignStmt
```

Fig. 2. Extracts of profiles produced when the PicoJava name analyser processes the program in Figure 1: attribute dimension (top); attribute and node type dimensions (bottom).

It is not possible to completely ignore tree traversal. When we are developing and debugging the equations, we need help to understand how they function. It is not enough to just look at each equation by itself, since the effect is achieved by a combination of many equations. Having some information about what happens at run-time can reveal much about how the attribute grammar works. As a simple example, if we knew that our name analyser never evaluated the `superClass` attribute when processing the program in Figure 1, we would know that the equations are not correctly implementing our intent.

2.5 Profiling PicoJava name analysis

The rest of this paper describes a method for producing run-time profiles of the execution of attribute evaluators. Before we present the detail, we give a couple of examples to illustrate useful profiles of the PicoJava name analysis attribute grammar.

The simplest profile we can imagine is one that shows us which attributes are evaluated during a run. The top profile in Figure 2 shows an attribute profile of the PicoJava name analyser that was produced as it processed the program in Figure 1.[1] The first part of the profile gives the total run-time and the time that is accounted for by the profiled attributes. 206 attribute instances were evaluated in this run. The table summarises the run-time by apportioning it to the attributes. Each row shows the total time taken by the evaluation of the attribute and the portions attributable to the equations of the attribute itself (`Self`) and of the attributes that those equations used (`Desc` for “descendants”). The final data columns show the number of times each attribute was evaluated.

The profile shows that the `decl` attribute and the attributes it uses consume the vast majority of the time, closely followed by `lookup` and `localLookup`. The profile reveals a number of areas where further investigation might be warranted. The `localLookup` attribute consumes almost a third of the time which seems excessive. We might investigate whether replacing a linear search by a hashed lookup would improve performance. Also, the `unknownDecl` attribute is used to return a special object to represent the case where a declaration cannot be found. It is worrying that computing this special object consumes ten percent of the time.

The top profile of Figure 2 shows just a single profile dimension: the attribute that was evaluated. Our profiles can summarise execution across more than one dimension to reveal more detail. For example, we might want to know the types of the nodes at which the `lookup` attribute was evaluated. The bottom profile in Figure 2 shows the `lookup` part of a multi-dimensional profile using the attribute and node type dimensions. In this table the rows summarise a particular combination of the `lookup` attribute and node type. For example, the first line summarises the cases where the `lookup` attribute was evaluated at `Use` nodes. Since the tree in Figure 1 contains three `Dot` nodes, it is comforting to see from the fifth line of the table that we looked up names three times at such nodes.

Profiles such as those shown in Figure 2 reveal a lot about the execution of an attribute evaluator in a form that is easy to absorb. Varying our choice of dimension means that we can tailor the profiles to the particular investigation that we are carrying out. We now consider how these sorts of profiles are produced in detail, before looking at more complex examples.

3 Attribute Grammar Profiling

Our approach to producing profiles such as those shown in the last section is to instrument programs to generate *domain-specific events* while they run. The events are grouped to form a *record-based model* of the execution in domain-specific terms. Reports are generated from the model by grouping records along developer-specified *dimensions*. We consider only profiling for attribute grammars in this paper, but the method is general. By varying the events that are generated and the dimensions that are available, profilers can be built for any domain.

Section 3.1 describes the data collection method and the record-based model of execution. Section 3.2 explains how the model is used to produce reports. The imple-

[1] Elapsed time is collected in nanosecond units, but is presented in the profiles as milliseconds, so there may be some rounding errors.

mentation of the approach for our Kiama language processing library is discussed in Section 3.3 and its performance is analysed in Section 3.4.

3.1 Data collection and execution modelling

The data collection approach is based on a simple event model. We distinguish between `Start` events that signal the beginning of some program activity and `Finish` events that signal the end of an activity. We assume that the program can be modified so that `Start` and `Finish` events will be generated at appropriate times.

Each event captures the event kind (`Start` or `Finish`), the time at which it occurred, the domain-specific type of the event, and a collection of arbitrary data items associated with the event instance. Each data item is tagged with a unique dimension. The dimensions that an event has at generation time are its *intrinsic dimensions*, to differentiate them from *derived dimensions* that are calculated later.

In the attribute grammar case, a single *attribute evaluated* event type is sufficient. We assume that a `Start` instance of this event type is generated just before the evaluation of an attribute begins, and that a corresponding `Finish` instance is generated just after evaluation of an attribute ends. Attribute evaluation events have the following intrinsic dimensions:

– *attribute*: the attribute that was evaluated,
– *subject*: the node at which the attribute was evaluated,
– *parameter*: the value of the attribute's parameter (if any),
– *value*: the value that was calculated by the attribute's equations, and
– *cached*: whether the value was calculated or came from the attribute's cache.

The attribute, subject and parameter dimensions must be present in both the `Start` and `Finish` events. We call this subset of the intrinsic dimensions the *identity dimensions*. They are used by the profiler to recognise when a `Finish` event matches a `Start` event that was seen earlier. The value and cached dimensions are present only in the `Finish` event since their values are available only after the evaluation has been completed.

After the execution is completed, we collect the events to create a list of *profile records* that describe the execution. When we see a `Finish` event we match it with the corresponding `Start` event by comparing the identity dimensions. Each matching `Start-Finish` pair is combined to form a single profile record. A record contains the event type, the time taken between the occurrence of the `Start` event and the occurrence of the `Finish` event, and the union of all of the intrinsic dimensions of the two events.

We also require that the events are generated in a last-in-first-out manner so that we can automatically construct a hierarchical model of execution. In other words, if we see a `Finish` event, it must be the case that the most recently seen `Start` event that does not yet have a corresponding `Finish` event is the one that corresponds to the new `Finish` event. If this condition holds, we can regard the records that are created between a `Start` event and its corresponding `Finish` event as the *descendants* of the new profile record. This hierarchical relationship is used to derive dimensions that relate records to each other to summarise attribute dependencies (Section 4.4).

Kind	*When*	*Attribute*	*Subject*	*Param*	*Value*	*Cached*
`Start`	4	`decl`	Node 1			
`Start`	4	`lookup`	Node 1	“a”		
`Start`	5	`lookup`	Node 2	“a”		
`Finish`	6	`lookup`	Node 2	“a”	Node 3	false
`Finish`	7	`lookup`	Node 1	“a”	Node 3	false
`Finish`	10	`decl`	Node 1		Node 3	false
`Start`	14	`decl`	Node 1			
`Finish`	15	`decl`	Node 1		Node 3	true

Record	*Time*	*Attribute*	*Subject*	*Param*	*Value*	*Cached*	*Descendants*
1	1	`lookup`	Node 2	“a”	Node 3	false	
2	3	`lookup`	Node 1	“a”	Node 3	false	1
3	6	`decl`	Node 1		Node 3	false	2
4	1	`decl`	Node 1		Node 3	true	

Fig. 3. Start and finish events from a program run (top) and the profile records that summarise the attribute evaluations signalled by the events (bottom).

To make this discussion more concrete, consider the execution of the PicoJava name analysis attribute evaluator. Among the events generated by this evaluator will be some that document the evaluation of the `decl` and `lookup` attributes. A possible execution results in the events shown in the top table of Figure 3. This trace excerpt describes four attribute evaluations. The first `Start` event marks the start at time step four of the evaluation of the `decl` attribute at Node 1. That evaluation requires an evaluation of the `lookup` attribute with parameter `"a"` at Node 1, and again at Node 2, which yields a value of Node 3, which we assume is the declaration node for `a`. The first evaluation of the `decl` attribute at Node 1 finishes at time step ten and did not use the `decl` attribute cache. If the value of the `decl` attribute at Node 1 is subsequently demanded again, represented by the final two events, its value will be obtained much more quickly using the attribute cache.

Four profile records will be created, one for each attribute evaluation (bottom table of Figure 3). For example, record three tells us that the first evaluation of `decl` took a total of six time units and required the evaluation represented by record two. Record two in turn took three time units and required the evaluation represented by record one.

3.2 Report generation

The record list produced by the data collection process is a domain-specific model of the execution. A report is produced from this model on the basis of one or more *report dimensions*. The report generation process proceeds by considering the records one-by-one and allocating them to *report buckets* according to their report dimension values. All of the records with the same report dimension values end up in the same bucket and their execution time is accumulated. The descendant information allows us to allocate the elapsed time to either the attribute evaluation represented by a record (self) or to the other evaluations demanded by that evaluation (descendants). When all of the buckets have been assembled, a table is printed where the rows represent the different buckets and are sorted in decreasing order or execution time.

For example, if we choose the attribute dimension and summarise the data from the records in Figure 3, we get the following profile report.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
    7 100.0     3  42.9     4  57.1     2  50.0  decl
    4  57.1     4  57.1     0   0.0     2  50.0  lookup
```

Since we are using the attribute dimension we get two buckets, one for each of the two attributes that we have recorded.

Multi-dimensional reports are produced by an analogous process. We first report on the basis of the first dimension. Then each bucket from the first dimension table is further summarised according to the second dimension. The process continues until there are no more dimensions to consider.

3.3 Implementation

The profiles presented in this paper were collected using an implementation we built as a library in the Scala language. Profiles can be generated from code written in any Java Virtual Machine language but specific support is provided for Scala and Java.

We manually instrumented our Kiama language processing library [23, 24], which is written in Scala, to use the profiling library to collect information about attribute evaluations. We augmented the attribute grammar evaluation code of Kiama to generate events as described above. While the new code had to be manually inserted, its total size is very small: only ten new method calls are needed in a module of more than five hundred lines. These calls produce the `start` and `finish` events. Their parameters are the event dimension names and values. This profiling support will be part of an upcoming release of our Kiama language processing library.

Users of Kiama need to only add one call to run their code under profiler control and to generate a report when their code is finished. All other instrumentation is in Kiama so the application code is not further affected.

An important implementation decision we made was to not encode the available dimensions and the types of their values into the profiling framework. A dimension is just represented by a string and a dimension value can be anything. Including more type information would enable a greater level of safety in the event generation and recording code. However, it would tie the framework to particular events and their dimensions. Adding new ones would require updating the interfaces or a more complex event representation with generic access to typed dimension values. Our approach is much simpler and is extremely flexible. Instrumentation code is one line at each event generation site. New events and dimensions can be added without recompiling the profiler.

Around 100 lines of support code in Kiama implements derived dimensions and access to the names of the attributes. In fact, the last of these is the only non-trivial part of the implementation. Unfortunately, Scala does not have full reflection yet so we are not able to easily recover the attribute names. As a work-around, we currently require the attributes to be defined as lazy values and some relatively fragile code examines the run-time stack to determine the names when the attributes are constructed. We plan to replace this code with a more robust implementation and allow non-lazy attribute definitions when proper reflection is available in the 2.10 release of Scala.

A profiling library component of less than 300 lines of code implements event capture, record representation, and report generation. The current implementation stores profile records as instances of a custom class. A generic format such as XML or JSON could easily be used instead and might be beneficial if the data was to be exported. As it stands, the event data is only used internally by the library, so this generality is not needed. We have not used any form of compression to reduce the space needed to store the profile records, since it has not been necessary for the test cases we have tried. It is possible that a need for space optimisation will be found when very large attribute grammars are profiled. If space usage becomes a serious problem, an on-the-fly approach could be the solution, where aggregation is performed as events are generated rather than at the end of the execution.

Events are time-stamped using the JVM `java.lang.System.nanoTime` method which has nanosecond precision. As in all profiling systems, the measured times vary from run to run depending on the machine load, but the relative times are stable. Precise nanosecond times are unlikely to be very useful since they present too much detail, so the profiler reports times in millisecond units.

Profiles that use intrinsic dimensions can be generated directly from the profile records. Derived dimensions can be added by overriding the default implementation of a library method that looks up dimension values. The method is given the dimension name and a reference to the profile record. The default implementation simply looks up the name in the record's dimension collection. An overriding implementation can return any value it likes. Sections 4.3 and 4.4 contain examples of derived dimension profiles.

The display of aggregated values can also be customised. By default, the profiler uses the standard Java `toString` method to obtain a string representation of a value. That implementation can be replaced by arbitrary code. For example, we could display subject trees using a pretty-printer instead of using the default representation. Since these sorts of values can take up a significant amount of space, they are unlikely to fit in the profile report tables. The report writer automatically detects when the value strings will not fit. Each such value is allocated a reference number which is used in the table. The actual value string is printed with the reference number below the table.

3.4 Performance

The performance of a profiling library is not important in production, but can make a difference to the efficiency of the development process. To explore the performance of our implementation, we conducted some experiments using an attribute grammar that is much bigger than the PicoJava one. The Kiama Oberon-0 example was developed for the tool challenge associated with the 2011 Workshop on Language Descriptions, Tools and Applications. Oberon-0 is the imperative language subset of the Oberon family of languages and was originally described by Wirth [28]. The challenge compiler parses and analyses Oberon-0 programs, then translates correct ones into equivalent pretty-printed C code. Our test case was compiling all of our fifty-two Oberon-0 test programs in a single run of the compiler. None of these programs is very large, but the compiler performs more than 32,000 evaluations of fourteen attributes while processing these files, so it is a serious test of the profiling system.

Running the Oberon-0 compiler on this test with profiling completely disabled takes about five to six seconds of elapsed time. Adding the event generation code to the library, but still with no report generation, doesn't make a difference that is noticeable when running from the command line. We also modified the program to collect the run-time for just the core compiler driver, thereby removing the time for other operations such as class loading. We ran all of the Oberon-0 tests in a single run as before, repeating the run ten times initially to warm up the virtual machine. Then we ran twenty-four tests, discarded both the slowest two and the fastest two results to remove any outliers, and averaged over the remaining twenty measurements. The results showed that the event generation by itself slows the core of the compiler down by a factor of about 1.4. While this is a significant difference, the command-line experiment shows that the slowdown is swamped by the time taken by other operations performed by the program. Thus, we believe that the instrumentation is practical for gathering data from large test runs.

We also investigated the time taken to produce profile reports. Producing a profile for the attribute dimension increases the run-time from around eight seconds with profiling turned on but no report generation, to about twelve seconds with report generation as well. Adding a second subject dimension increases the total time to over twenty-two seconds. Most of this time is taken by printing the many tree fragments, which illustrates that report generation time is highly dependent on the chosen dimensions. We have not performed any optimisation of the core of the report generator so it is likely that some improvement could be obtained. Nevertheless, our experiments show that the current performance is practical for typical interactive uses during development.

4 More complex profiles

To further demonstrate the power of our profiling approach and to provide a context for more detailed discussion, we conclude the core of the paper by considering more complex profiles. In particular, we consider different dimensions that improve our ability to understand the execution of our attribute evaluators. The profiles are produced from executions of the PicoJava and Oberon-0 compilers.

4.1 Parameterised attributes

Parameterised attributes are a powerful feature of modern attribute grammar systems, but their operation can be opaque since the parameter values are not part of the equations. Seeing the parameter values that are used can greatly increase understanding and reveal problems. A profile along the attribute and parameter dimensions shows us exactly which parameters are being used and how often. For example, the PicoJava name analyser yields the following profile for the attribute `localLookup`. Parameter values are optional, so they are shown as either `Some(v)`, representing the value `v`, or `None`, which denotes that a parameter was not used.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
    8  22.1     4  11.2     4  11.0     5   2.4  Some(int)
    3   9.7     2   5.9     1   3.8     1   0.5  Some($unknown)
    0   1.4     0   1.0     0   0.4     3   1.5  Some(BB)
    0   0.8     0   0.7     0   0.1     3   1.5  Some(y)
    0   0.7     0   0.6     0   0.1     3   1.5  Some(x)
    0   0.5     0   0.5     0   0.0     2   1.0  Some(AA)
    0   0.2     0   0.2     0   0.0     1   0.5  Some(a)
    0   0.2     0   0.2     0   0.0     1   0.5  Some(b)
```

Quite a lot of time is spent looking up the pre-defined type name `int` and something called `$unknown`. The latter is only looked up once and still manages to take more time than the lookups of "real" names. An examination of the source code triggered by this profile reveals that `$unknown` is the name given to the unknown declaration. As we observed in Section 2.5, some improvements could be made to the handling of unknown declarations and apparently this profile shows another symptom. We would also hope to reduce the time spent looking up pre-defined names.

4.2 Structured attributes

Kiama has *structured attributes* that are groups of attributes that are associated with each other in some way. One kind of structured attribute is a *chain*, which is inspired by a similar construct in the LIGA attribute grammar system [13]. A chain abstracts a pattern of attribution that threads a value in a depth-first left-to-right fashion throughout a tree. The idea is that the system provides the default threading behaviour and the attribute grammar writer can customise the equations at various places in the tree to update the chain value. Kiama chains are implemented by a pair of attributes: one to calculate the value of the chain that comes in to a node from its parent, and one to calculate the value that goes back out of the sub-tree to the parent. The developer can provide functions to transform the incoming value as it heads into a sub-tree or as it leaves the sub-tree.

The Oberon-0 attribute grammar uses a chain to encode an environment that propagates symbol information from declarations to uses. Declarations add information to the chain and uses of names access the chain to look up information. This approach contrasts with the name analysis approach used in the PicoJava example where parameterised attributes are used to search for the declaration information to bring it to the uses where it is needed. Profiles can help us compare these two approaches to this problem. We can profile the PicoJava name analyser along the attribute and parameter dimensions to assess the overhead of looking up each name individually. We can profile the environment attribute in the Oberon-0 compiler to find out the costs of propagating it.

4.3 Derived dimensions

All of the profiles we have seen so far use the intrinsic dimensions whose values are already present in profile records. A powerful extension is to produce profiles based on dimensions that are derived from the profile records. Derived dimensions summarise the execution in any way we like, including in ways that are specific to the application.

For example, suppose that we are interested in knowing where the attribute evaluations occur within the tree. We might wish to know whether particular attributes are evaluated more at boundary nodes (the root and the leaves) or at the other (inner) nodes. The profile records do not have an intrinsic dimension that gives us this information directly, but we can calculate this new location dimension from the subject dimension.

A simple implementation of the location dimension suffices to return one of the strings "Root", "Inner" or "Leaf", depending on the location of the subject tree node within the tree. Here is a typical profile that results from using the attribute dimension with this new location dimension. We show the profile for the input half of the environment chain from the Oberon-0 attribute grammar.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
  185  26.5    52   7.5   132  18.9  3237  10.0  Leaf
  133  19.0    44   6.3    89  12.8  5455  16.8  Inner
   58   8.3    58   8.3     0   0.0    81   0.3  Root
```

The profile shows that this particular attribute is evaluated quite a lot at the leaves, which we would expect since the names reside at the leaves and attribution associated with them is the primary client of the environment. Evaluation at inner nodes is necessary to transport the environment from the declarations where it is established to the leaves. The many evaluations at the root are more of a mystery, since we would expect to only determine an incoming environment at the root once for each input program, since that value just contains pre-defined names. (Recall that there are fifty-two programs in the full Oberon-0 test suite.) Examination of the compiler implementation shows that attributed trees are transformed and the transformed trees are sometimes re-attributed, so this repeated evaluation at a root location also makes sense. Some of the programs only have one root evaluation as they are erroneous and are not translated.

Location is a very simple derived dimension but the idea can be taken as far as necessary to reveal just those aspects of the execution needed to diagnose a problem or to expose the way that some attribution works. A particularly powerful application of derived dimensions is custom views of attribute values. We can create dimensions to show the values aggregated according to any useful criteria. For example, an attribute might hold a pretty-printed version of some part of the tree. We could aggregate evaluations of that attribute along a dimension that calculates the number of lines in the pretty-printed text. A profile along this derived dimension would allow us to analyse the impact of the complexity of the pretty-printing task on the run-time.

4.4 Attribute dependencies

It is sometimes desirable to examine the dynamic dependencies that are induced by the application of the attribute equations to a particular input. Dynamic dependencies reveal exactly how the attributes depend on each other in a particular execution. In contrast, static dependencies that can be obtained by examining the equations themselves are an over-approximation of the dependencies that are actually used.

Kiama's *depends-on* derived dimension aggregates attribute evaluations according to the attributes on which they directly depend.[2] For example, here is a profile along the attribute and depends-on dimensions for the idntype attribute from the Oberon-0 compiler. The idntype attribute holds the type of an identifier use. (We omit the timings from this table and just show the bucket counts to save space.)

```
754   2.3  IdnUse(@).entity, NamedType(?).deftype
155   0.5  IdnUse(@).entity
 13   0.0  IdnUse(@).entity, IntExp(?).tipe
 22   0.1  IdnUse(@).entity, RecordTypeDef(?).deftype
  4   0.0  IdnUse(@).entity, AddExp(?).tipe
 24   0.1  IdnUse(@).entity, ArrayTypeDef(?).deftype
  1   0.0  IdnUse(@).entity, DivExp(?).tipe
  1   0.0  IdnUse(@).entity, IdnExp(?).tipe
```

Each row of this table reports a particular pattern of dependence. For instance, the first line says that in 754 cases the attribute depended on the entity attribute of an IdnUse node and the deftype attribute of a NamedType node. The parenthesised characters after the node types indicate where the given node was in the tree relative to attribute evaluation node. An at-sign means at the same node and a question mark means at a node that is not directly connected to the attribute evaluation node.[3]

The profile reveals a number of things about the calculation of the idntype attribute. Firstly, each case involves the entity attribute of the same node, which makes sense since the entity is to what the identifier refers. We find the type in different ways depending on the kind of entity the name represents. Secondly, on 155 occasions the attribute does not depend on any other attributes because the type can be obtained directly from the entity. In the AddExp, DivExp and IdnExp cases the type is obtained from an expression, which will happen if the named entity is a defined constant. Finally, if the name refers to some other kind of entity, its type will be determined from the type of its declaration which will be a ArrayTypeDef.

Profiles of this kind are particularly useful for diagnosing issues with the distribution of test cases. They are a convenient way to gather statistics about how many instances of particular circumstances occur in a test run, since only some kinds of expression are present. For example, from the profile above we might conclude that more test cases are required to ensure that every possible kind of constant expression is tested. If the language supports type definitions other than type aliases, record, and array types, we should add tests for the other cases, since they are missing from the profile.

A dependency profile can be used with node location profiles (Section 4.3) to summarise the pattern of dependence of an attribute with a view to simplifying that attribute's equations. For example, suppose we find that an attribute only depends on

[2] Another dependencies dimension considers all transitive dependencies and generates visualisations for display by GraphViz.

[3] depends-on profiles can also indicate references to the parent node, to children nodes, or to the next or previous nodes in a sequence.

itself at its parent node or on nothing at the root. This pattern of dependence is common and can be abstracted out using an attribute decorator [14]. A `down` decorator takes care of the transport of the attribute value down the tree from the root to where it is needed. The developer need only explicitly state the computation that should happen at the root. Thus, profiling can assist with refactoring of attribute grammars into simpler forms.

5 Discussion and Related Work

The vast majority of previous work in the profiling area has been concentrated on execution profiles for programs, inspired by systems that include the seminal `gprof` tool [6]. From the perspective of our work, `gprof` and descendant tools collect events that correspond to function calls. The call graph of the program corresponds to our descendant relationship between profile records. Our profile reports are inspired by those of `gprof`. We also distinguish between the time taken by an evaluation and those that it uses, as `gprof` distinguishes between the time taken by a the caller and the callees.

Our approach is more general, since `gprof` and similar tools solely use the function dimension, whereas we show the utility of domain-specific and derived dimensions to provide different views of the execution. `gprof` focuses on execution time and function call counts so issues of efficiency are paramount. In contrast, our flexible dimension-based approach means that the values themselves are often more important than times.

Abstraction in profiling systems. Instead of profiling at the function level, our approach raises the level of abstraction. Abstraction increases the generality of the profiling system and significantly reduces the size of the collected data compared to instruction and function-level profilers.

Sansom and Peyton Jones describe a profiler for higher-order functional languages including Haskell [20]. The execution of higher-order programs, particularly lazy ones, is not obvious since the compilation process is non-trivial and execution order often does not correspond clearly to the source code. They allow developers to add “cost centres” that aggregate data in a program-specific manner. Thus, the source-level profile data can be lifted to a higher level. In our case, the data is always at a higher level. Their cost centres are each associated with source code fragments rather than separated into `Start` and `Finish` events as in our approach. Thus, the identification of an abstracted piece of program execution is more flexible in our approach because the two events do not have to be associated with the same source code.

Nguyen *et al.* describe a domain-specific language for automating the regulation of profile data collection, processing and feedback [17]. The language allows some abstraction away from the details of the profile exploration process. However, it is different from our work in that it operates at a low level and is intended for analysing performance of programs and system kernels in a similar fashion to `gprof`.

Rajagoplan *et al.* consider profiling for event-based programs such as graphical user interfaces [19]. Events in their work are intrinsic to the functioning of the system, whereas in our work they are solely part of the profiling system. They look for patterns in the events which allows them to abstract away from the execution somewhat, but they do not consider a general abstraction mechanism.

Systems such as DTrace [16], the Linux Trace Toolkit [29] and the Java Virtual Machine Tool Interface can be used to collect a large amount of trace data about program execution. Some customisation of the data collection is usually possible. For example, DTrace allows the developer to write custom probes that are inserted and executed efficiently by the infrastructure. A general event tracing mechanism of this kind could be used to collect data about attribute evaluation. However, these approaches are quite heavyweight and have a great deal of machine or system dependence to achieve efficiency. Our work shows that a lightweight, simple approach is sufficient for monitoring the execution of language processors.

Bergel *et al.* describe a model-based domain-specific profiling approach [1]. Instrumentation code augments domain model code via an existing event mechanism or by using the host language's reflection framework. Custom profilers use the information gathered to display profiles and visualisations that relate directly to domain data and operations. The reliance on meta-programming features of the framework distinguishes their approach from ours. We pay the price of having to instrument the attribute grammar library code by hand, but as a result we make no demands on the underlying runtime. Our profile library is independent of any particular domain and the reports have a generic format, whereas their profilers are deliberately tailored to particular model code and particular kinds of observations that they want to make.

Profiling attribute grammars. Saraiva and colleagues have investigated the efficiency of attribute grammar evaluation approaches, notably as part of work to improve the efficiency of evaluators that are constructed as circular programs [5]. Their experiments focused on course-grained measures such as heap usage, rather than the kind of fine-grained analysis considered in this paper. In earlier work, Saraiva compared the performance of functional attribute evaluators [21]. He examined properties such as hash table size, cache misses and the number of equality tests performed between terms for both full and incremental evaluation of attributes. Some of these measures have analogues in our approach. The implementation appears to be custom to the particular experiment rather than a general facility as in our library.

Söderberg and Hedin show how attribute profiles can be used to analyse caching behaviour in JastAdd [25]. They calculate an *attribute instance graph* that is an attribute dependence graph with evaluation counts on the edges. Edges labelled with counts greater than one point to attributes for which caching might be advantageous. Their attribute dependence graph is similar to our collection of profile records and dependency relationships, except that our records are independent of the attribute evaluation domain. Our use of arbitrary dimensions to extend the power of profiles goes beyond the aim of Söderberg and Hedin's work which was to look solely at caching issues.

We developed the Noosa execution monitoring system for the Eli system, including the attribute grammar component [22]. Noosa is a debugging system, not a profiler, but it also uses an event-based approach to record information about the execution of a program. Noosa doesn't group events in the same way as the Kiama profiler, since it uses events primarily to specify domain-specific breakpoints. The focus is on controlling the execution as it happens rather than on summarising it after it is done. Noosa can be used to examine the values of attributes of interest with reference to the abstract syntax tree, but it cannot be used to summarise the execution along other dimensions.

6 Conclusion and Future Work

We have described a new general approach to domain-specific profiling and its application to profiling dynamically-scheduled attribute grammar evaluators. The approach is easy to implement and its execution overhead is low enough for interactive use. We have described applications of the profiler to attribute grammar understanding, test case coverage, and refactoring.

Our current implementation does not handle Kiama's *circular attributes* that are evaluated until they reach a fixed point. We plan to add support for circular attributes as we have described for regular and parameterised ones. We will also add dimensions that enable the operation of the fixed point computation to be examined. For example, a useful dimension would be the number of times that a circular attribute had to be evaluated before it reached a fixed point. It would also be useful to be able to distinguish the run-time devoted to each iteration in the computation of a circular attribute.

One direction of future work will be to deploy the profiler for use in other attribute grammar systems. The approach used with Kiama should easily transfer to other systems based on dynamic scheduling, but minor modifications should allow it to be used with other evaluation approaches. For example, a statically-scheduled tree walking attribute evaluator could be instrumented automatically by the scheduler. There might be scope to add new dimensions, such as one that captures information about node visits.

We also plan to use our framework to investigate applications of profiling other aspects of language processing. A student is also working on a user interface for interactive and graphical access to the profiling data.

References

1. Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-specific profiling. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2011.
2. Béatrice Bouchou, Mirian Halfeld Ferrari, and Maria Lima. Attribute grammar for XML integrity constraint validation. In *Database and Expert Systems Applications*, volume 6860 of *Lecture Notes in Computer Science*, pages 94–109. Springer, 2011.
3. Drew Davidson, Randy Smith, Nic Doyle, and Somesh Jha. Protocol normalization using attribute grammars. In *Computer Security – ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2009.
4. Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 1–18, New York, NY, USA, 2007. ACM.
5. João Paulo Fernandes, João Saraiva, Daniel Seidel, and Janis Voigtländer. Strictification of circular programs. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 131–140, New York, 2011. ACM.
6. Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.
7. Feng Han and Song-Chun Zhu. Bottom-up/top-down image parsing with attribute grammar. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(1):59 –73, 2009.
8. Görel Hedin. Reference Attributed Grammars. *Informatica*, 24(3):301–317, 2000.

9. Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
10. Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *Proceedings of the International Symposium on Programming*, pages 167–178. Springer, 1984.
11. Muhammad Karim and Conor Ryan. A new approach to solving 0-1 multiconstraint knapsack problems using attribute grammar with lookahead. In *Genetic Programming*, volume 6621 of *Lecture Notes in Computer Science*, pages 250–261. Springer, 2011.
12. Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.
13. Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
14. Lennart Kats, Anthony M. Sloane, and Eelco Visser. 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, pages 142–157. Springer, 2009.
15. Eva Magnusson and Görel Hedin. Circular reference attributed grammars–their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.
16. J. Mauro, B. Gregg, and C. Mynhier. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.
17. Peter Nguyen, Katrina Falkner, Henry Detmold, and David S. Munro. A domain specific language for execution profiling & regulation. In *Proceedings of the Australasian Conference on Computer Science*, pages 123–132. Australian Computer Society, Inc., 2009.
18. Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *Computing Surveys*, 27(2):196–255, 1995.
19. Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. Profile-directed optimization of event-based programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 106–116, New York, NY, USA, 2002. ACM.
20. Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385, March 1997.
21. João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.
22. Anthony M. Sloane. Debugging Eli-generated compilers with Noosa. In *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 1999.
23. Anthony M. Sloane. Lightweight language processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 2011.
24. Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure embedding of attribute grammars. *Science of Computer Programming*, In press, 2012.
25. Emma Söderberg and Görel Hedin. Automated selective caching for reference attribute grammars. In *Proceedings of the 3rd International Conference on Software Language Engineering*, pages 2–21, 2010.
26. Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1+2):39–54, January 2010.
27. H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 131–145. ACM Press, 1989.
28. Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996. Revised Nov. 2005.
29. Karim Yaghmour and Michel Dagenais. The Linux trace toolkit. *Linux Journal*, May 2000.