A Formalisation of Parameterised Reference Attribute Grammars

Scott J. H. Buckley and Anthony M. Sloane
Programming Languages and Verification Research Group
Department of Computing, Macquarie University, Sydney, Australia
{Scott.Buckley,Anthony.Sloane}@mq.edu.au

Abstract

The similarities and differences between attribute grammar systems are obscured by their implementations. A formalism that captures the essence of such systems would allow for equivalence, correctness, and other analyses to be formally framed and proven. We present Saiga, a core language and small-step operational semantics that precisely captures the fundamental concepts of the specification and execution of parameterised reference attribute grammars. We demonstrate the utility of Saiga by a) proving a meta-theoretic property about attribute caching, and b) by specifying two attribute grammars for a realistic name analysis problem and proving that they are equivalent. The language, semantics and associated tests have been mechanised in Coq; we are currently mechanising the proofs.

CCS Concepts • **Software and its engineering → Semantics**;

Keywords attribute grammars, small-step operational semantics, name analysis

ACM Reference Format:
Scott J. H. Buckley and Anthony M. Sloane. 2017. A Formalisation of Parameterised Reference Attribute Grammars. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136024

1 Introduction

Attribute grammars are a well-studied specification approach for tree-based analysis, particularly for static analysis of programming languages. Modern attribute grammar systems share many concepts but unfortunately system specifics obscure the essence of these concepts. For example, JastAdd [2], LRC [12], Kiama [14] and Silver [15] each implement a form of reference attribute grammars. However, these systems use different approaches to implement tree traversal and to allow attribute values to reference tree nodes.

If we wish to study attribute grammars from a formal standpoint, choosing any one system as the basis of that study risks confusing the essence of the concept with implementation detail. For example, we might be interested in meta-theoretic properties such as the behaviour of an attribute caching scheme. Or we might want to analyse particular attribute grammars, perhaps to discuss their efficiency or to compare them.

In each of these cases we are hampered if we choose a particular implementation that will introduce many details that are not relevant to the analysis at hand. For example, when considering the effect of caching on a computation it is not necessary to know exactly how that caching is implemented. When studying efficiency or comparing attribute grammars we don't want to compare actual run-times or low-level events that occur during execution of an implementation. Rather we want to study occurrences of domain-level events such as the evaluation of an attribute or the placement of a value in a cache, since their study reveals more generally applicable results.

A standard approach for this kind of situation is to define a core language that captures the concepts of interest. Type systems and operational semantics for the core language enable its properties to be studied independently of implementation complications.

In this paper we present Saiga, a core language that captures the main concepts of modern attribute grammar evaluation. Specifically, Saiga defines parameterised, reference attribute grammars and they are evaluated using dynamic scheduling and optional attribute caching. Section 2 presents the language and illustrates its use by way of a simple example inspired by name analysis.

Section 3 formalises the meaning of Saiga by way of a type system that defines well-formed programs and a small-step operational semantics that gives meaning to those programs. The full formalisation is reached in three stages: a base level that contains simple (reference) attributes and no caching, the addition of parameterised attributes, and the addition of caching.

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136024

Section 4 explores meta-theoretic properties of Saiga's formalisation. We begin with basic properties: *determinism*, *strong progress* and *type preservation*. Then we argue that the important property of *cache irrelevance* holds; i.e., that the presence or absence of caching does not affect the values computed for attributes. We also argue that cached-based evaluation performs no more work than an evaluation that doesn't use caching.

Section 5 demonstrates that Saiga is expressive enough to model real-world uses of attribute grammars in language engineering and that we can reason about these attribute grammars. We compare two attribute grammar specifications of name analysis for PicoJava, a cut-down Java containing only constructs that are relevant for name analysis. Our analysis proves that the specifications are equivalent in the sense that for any PicoJava program they compute the same defining identifier occurrence for each applied identifier occurrence.

Overall, our formalisation of attribute grammars meets the goal of allowing formal properties to be discussed and proven independently of the particular implementation choices of any attribute grammar system. The language, semantics and tests have been mechanised using the Coq proof assistant and the mechanisation can be found in the following repository:

https://bitbucket.org/scottbuckley/saigacoq

Related work: Knuth introduced attribute grammars via an abstract process of evaluating attribute equations "until no more attribute values can be defined" [7]. Since then, attribute grammar evaluation has often been described by a translation to an evaluator implementation written in another language. Conditions are often imposed on attribute equations to facilitate a feasible translation, such as rejecting circularly-defined equations. Researchers have defined translations to efficient evaluators. For example, Kastens defined the class of *ordered attribute grammars* for which it is possible to derive a tree-walking evaluator that works for any syntactically-correct tree without having to make any scheduling decisions at run-time [6]. Johnsson described how evaluation can be achieved easily in a lazy functional language [4]. Jourdan described a translation to recursive functions that eschews static checks, schedules dynamically and detects circularity at run-time [5].

These approaches and many others like them rely on evaluation of another language. Thus, formal reasoning about evaluation of an attribute grammar requires formal reasoning about that other language. For example, to understand the meaning or behaviour of a tree-walk evaluator derived from an ordered attribute grammar, we need a model of the tree-walking process. But this process necessarily contains details specific to that implementation approach, thereby complicating comparison with other approaches. The essence of the attribute grammar has been obscured by the tree walker.

We get into similar difficulties when we consider more modern attribute grammar features. To pick just one example from many, the description of reference attribute grammars by Hedin includes both a direct object-oriented implementation as well as a translation to non-reference attribute grammar constructs [3]. Neither of these is particularly useful for studying reference attributes from first principles since both translations lose sight of the fact that reference attributes are not really that different from non-reference attributes.

De Moor *et al.* showed how to define compositional attribute grammars in Haskell using an aspect-oriented approach [9, 10]. This paper is a successor to the earlier work on implementing attribute grammars as functional programs mentioned above and relies crucially on laziness. Similarly, Backhouse defined a Haskell-based implementation of attribute grammars that is used to reason in "a calculational style" and to derive a new test for definedness [1]. Schäfer *et al.* implemented circular, reference attribute grammars as a shallow embedding in Coq [13]. They used zippers to keep track of locations in trees (an approach that has been explored further by others [8]). They are able to prove properties of non-trivial attribute grammars, but overall the result is similar to implementing attribute grammars in a general-purpose functional language.

These shallow embedding approaches in functional languages are very powerful, particularly when we consider how to write attribute grammars concisely using components and how to reason about them equationally. However, the essence of attribute evaluation is quite obscured since we are very quickly transported to the level of Haskell, Coq, or similar.

Our aim is to examine the essence of attribute evaluation, which we achieve by defining a new core language that abstracts away from other details such as high-level specification notations. We define the core language's semantics directly by inference rules and by mechanising those rules via a deep embedding in Coq. The result is a small-step version of Jourdan's dynamically scheduled evaluation approach [5] which is used by modern attribute grammar systems and libraries, including JastAdd [2], Kiama [14] and Silver [15].

The semantics utilises an underlying mechanism to evaluate functions on attribute values but this evaluation is standard and lies outside the rules that define how attribute evaluation proceeds, how results are cached, and so on. This separation means that we can reason about the evaluation aspects that matter while keeping the semantics simple by deferring unrelated details to the functional layer.

We shift questions of attribute grammar specification to a general "context" function that allows us to abstract away the differences between specification approaches, thereby unifying attribute definitions, tree structure and caching. In contrast to standard treatments of attribute grammar evaluation, our approach unifies evaluation of synthesised and inherited attributes while easily handling reference attributes and parameterised attributes.

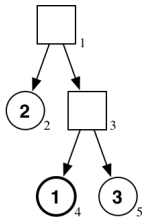

Figure 1. A tree with integer leaves and binary interior nodes. Node four is the tree's minimum leaf.

e	::=	v	value
	|	`IF` e_1 `THEN` e_2 `ELSE` e_3	conditional
	|	$e_1(e_2)$	function apply
	|	$e.a$	attribute value

Figure 2. Abstract syntax of the Saiga language.

2 A Core Attribute Grammar Language

Consider trees that contain integer leaves and binary interior nodes such as in Figure 1. Suppose we wish to identify a *minimum leaf* (a leaf that contains the minimum value of any leaf in the tree) and to communicate a reference to that minimum leaf to each leaf. In the example, the minimum leaf is node four. This problem is similar to the name analysis problem for programming languages where the sought-after nodes represent defining occurrences of identifiers and the destinations represent applied occurrences. We discuss a fuller version of name analysis in Section 5.

We can decompose this problem into two sub-problems: 1) identify the minimum leaf in a sub-tree rooted at a particular node, and 2) communicate the minimum leaf of the whole tree to each leaf.

Problem 1 has two cases: a) if the sub-tree root is a leaf then the minimum leaf is that leaf node, and b) otherwise, the node is a binary interior node and we pick the minimum leaf of the children that has the smaller leaf value. If two children have the same minimum leaf value we arbitrarily choose the left child's minimum leaf. In attribute grammar terms, Problem 1 can be solved with a single synthesised reference attribute that propagates the appropriate minimum leaf reference up the tree making choices at each binary node.

Problem 2 is a simple propagation of the minimum leaf of the root of the tree down to each leaf. This computation can be achieved using a single inherited attribute.

2.1 Core Language Expressions

Attribute grammar computation notations differ considerably from system to system. We use a simple expression language to abstract from those details (Figure 2). Computations can be a basic value (v), a conditional expression (`IF` e_1 `THEN` e_2 `ELSE` e_3), an application of a function ($e_1(e_2)$), or a use of an attribute value ($e.a$). In Section 3 a type system will ensure that expressions make sense. For example, in an attribute value $e.a$ the expression e must evaluate to a node reference. For now we assume type correctness.

The expression forms are designed to capture necessary basic computational elements but not to over-specify them. Conditionals are included so that a computation can elect not to evaluate some sub-expression (i.e., conditionals are the only lazily-evaluated construct). Basic values include functions for use in application expressions. The exact choice of basic values and functions over those values is left unspecified since it will depend on the particular problem and does not affect how attributes are evaluated.

2.2 The Context

Attribute grammar systems have widely varying notations for specifying the attribute computation to perform to calculate the value of a particular attribute occurrence. Some systems associate attribute equations with grammar symbols, others with grammar productions, and some use a mixture. Higher-level notations include reusable equation modules, inheritance of equations and forwarding specifications.

In our formalism we abstract away from the details of these different specification approaches using a single *context function* with the following type:

$$\sigma : Nodes \times Attributes \longrightarrow Expressions$$

$\sigma(n, a)$ is the expression to use to compute the value of attribute a of node n. (An extension is necessary to accommodate parameterised attributes; see Section 3.3. Caching can be accommodated by updating the context; see Section 3.4.)

The context abstraction also encodes the tree structure. That is, it naturally accommodates what we will call *intrinsic attributes* whose values are available directly from the tree, as opposed to *extrinsic attributes* whose values are determined via evaluation. We assume that for each intrinsic attribute the context returns an expression that is the intrinsic attribute value. Inter-node links such as between a node and its children or between a node and its parent are intrinsic attributes of node type.

2.3 Problem Solution

The problem described at the beginning of this section relies on the following intrinsic attributes:

- if n is a leaf node, $n.value$: an intrinsic attribute whose value is the integer stored in n,
- if n is a binary node, $n.childl$ and $n.childr$: references to the left and right child of n, respectively, and
- $n.parent$: a reference to the parent node of n, or *null* if no parent exists.[1]

[1] We should probably use an option type here but we omit this improvement to keep the presentation simple.

We assume that the context defines values for all intrinsic attributes.

We can solve the problem by defining the following attributes for all nodes n:

- $n\,.\,minleaf$: a reference to the minimum leaf node in the sub-tree rooted at n,
- $n\,.\,treeminleaf$: a reference to the minimum leaf node in the whole tree.

Defining these attributes amounts to specifying the expressions to be returned by the context function σ (Figure 3). In the leaf case *minleaf* simply returns the leaf itself. In the binary case it compares the values of the minimum leaf nodes of the children and chooses the leaf that has the smaller value. The function *le* is the standard less-than or equal function on numbers.

The definition of *treeminleaf* copies the *treeminleaf* value from the parent if there is a parent, otherwise it is at the root node so it uses the *minleaf* of the root. The parent test uses reference equality *equal*.

A key aspect of this formalisation approach is the distinction between the choice of expression to use as the definition for an attribute occurrence and the choices that attribute definitions themselves make. The former are part of the attribute grammar specification as modelled by the context and are exemplified in Figure 3 by the choice between equations (1) and (2) for *minleaf*. The latter are part of the computation carried out by the attribute grammar and are exemplified by the conditional expressions in equations (2) and (3).

2.4 Evaluation

Section 3 formally defines how to carry out an attribute computation defined using our approach. For now we sketch how evaluation works for our example.

Evaluation is triggered by a demand for the value of some attribute at some node. A typical example might be the need for type information about an applied identifier occurrence as the user's mouse hovers over it in an editor. Thus, we begin evaluation via an attribute of the form $n\,.\,a$. For the sake of the example we choose the *treeminleaf* attribute of node two in Figure 1. Throughout we denote node i by n_i, so our starting expression is $n_2\,.\,treeminleaf$.

Evaluation begins as follows where equation numbers refer to Figure 3.

$$\begin{array}{lll} & n_2\,.\,treeminleaf & (3) \\ \longrightarrow^* & n_2\,.\,parent\,.\,treeminleaf & \\ \longrightarrow^* & n_1\,.\,treeminleaf & (3) \\ \longrightarrow^* & n_1\,.\,minleaf & \end{array}$$

At this point we have reduced the original problem to one of determining the minimum leaf node for the root of the tree. Evaluation now proceeds downward using equation (2) until leaves are reached where we use equation (1). At each binary node we calculate the value of each child's minimum leaf and compare them. For example, to evaluate $n_1\,.\,minleaf$ we need

$$\begin{array}{lll} & n_1\,.\,childl\,.\,minleaf\,.\,value & \\ \longrightarrow^* & n_2\,.\,minleaf\,.\,value & (1) \\ \longrightarrow^* & n_2\,.\,value & \\ \longrightarrow^* & 2 & \end{array}$$

A similar computation of $n_1\,.\,childr\,.\,minleaf\,.\,value$ calculates $n_1\,.\,childr\,.\,minleaf$ to be n_4. Then $n_4\,.\,value$ is 1, so rule (2) at n_1 chooses $n_1\,.\,childr\,.\,minleaf$. Therefore the value of the attribute $n_1\,.\,minleaf$ is n_4. The first sequence of evaluation steps above means that the attributes $n_1\,.\,treeminleaf$ and $n_2\,.\,treeminleaf$ also have value n_4.

It is clear from this computation that caching attribute values can be useful. For example, there is no need for the computation of a node's *treeminleaf* attribute to always go all the way to the root of the tree via rule (3). It need only go up until it finds a *treeminleaf* that has already been computed. Thus, n_4 and n_5 can share $n_3\,.\,treeminleaf$.

3 Formalisation

In this section we make the notions from Section 2 precise. First, we define the type structure for attribute grammar expressions and the context. Then in three stages we define a small-step semantics for expression evaluation: first just for basic reference attributes, then adding parameterised attributes and then adding attribute value caching.

3.1 Types

Figure 4 summarises the meta-variables and syntactic categories of the Saiga language and its formalisation. Saiga expressions evaluate to values from any set of basic types $\boldsymbol{T}$ that includes at least the Boolean type, the types of functions between basic types, and the special type $\boldsymbol{N}$ that categorises tree node references.

$\boldsymbol{A}$ categorises attributes and τ gives the declared types of attributes. We assume a single global τ that is defined for all attributes. Attributes can be considered labels for data that is associated with nodes.

We use only one type of attribute because there is no need to distinguish between them for the purposes of evaluation using our semantics. For example, as we will see, intrinsic and extrinsic attributes only differ in the form that their definition takes: either a precomputed value or an expression that might require some further evaluation, respectively. Similarly, from the perspective of our evaluation scheme there is no difference between synthesised and inherited attributes. The same evaluation approach works for each kind. Indeed, a single attribute may have both aspects of synthesised evaluation (from below) and of inherited evaluation (from above) without causing a problem. Mapping attributes from any particular user-level attribute grammar language down to these more fundamental mechanisms is outside the scope of this paper.

$$\sigma(n, \mathit{minleaf}) \quad \text{if } n \text{ is a leaf node} \quad = \quad n \tag{1}$$

$$\text{if } n \text{ is a binary node} \quad = \quad \texttt{IF}\ \mathit{le}(n.\mathit{childl}.\mathit{minleaf}.\mathit{value})(n.\mathit{childr}.\mathit{minleaf}.\mathit{value}) \tag{2}$$
$$\texttt{THEN}\ n.\mathit{childl}.\mathit{minleaf}$$
$$\texttt{ELSE}\ n.\mathit{childr}.\mathit{minleaf}$$

$$\sigma(n, \mathit{treeminleaf}) \quad = \quad \texttt{IF}\ \mathit{equal}(n.\mathit{parent})(\mathit{null}) \tag{3}$$
$$\texttt{THEN}\ n.\mathit{minleaf}$$
$$\texttt{ELSE}\ n.\mathit{parent}.\mathit{treeminleaf}$$

Figure 3. Definitions of $\sigma(n, \mathit{minleaf})$ and $\sigma(n, \mathit{treeminleaf})$. *le* is the less-than or equal to operation on integers and *equal* is reference equality.

Further, we model tree structures in a generic way via intrinsic attributes, so no particular structure is assumed by the formalism. When a particular attribute grammar needs attributes that define the shape of a tree (such as the children and parent(s) of each node) they are just assumed intrinsic attributes. Thus, an operation such as accessing the children of a node is evaluated in the same way as accessing any other attribute of that node.

Contexts are the key abstractions that hold information about the structure, attribute values, and attribute equations. Formally, the context is a total map from nodes and attributes to expressions. Using a total map allows us to elide another aspect of attribute grammar specification: whether an attribute has a definition or not (which includes whether an intrinsic attribute has a value or not). In other words, we are assuming that completeness and well-definedness issues have been resolved before evaluation is attempted. We also assume that the initial context σ is correctly typed:

$$\tau(a) = t \implies \forall n \in \boldsymbol{N}.\sigma(n, a) \in t$$

Expression	$e \in \boldsymbol{E}$
Type	$t \in \boldsymbol{T}$
Node	$n \in \boldsymbol{N}$
Attribute	$a \in \boldsymbol{A}$
Attribute type	$\tau \in \boldsymbol{A} \to \boldsymbol{T}$
Context	$\sigma \in \boldsymbol{N} \times \boldsymbol{A} \to \boldsymbol{E}$

Figure 4. The meta-variables and syntactic categories of the Saiga language formalisation.

$$\frac{}{\mathit{true} \in \mathit{Bool}} \qquad \frac{}{\mathit{false} \in \mathit{Bool}}$$

$$\frac{e_1 \in \mathit{Bool} \qquad e_2 \in T \qquad e_3 \in T}{\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \in T}$$

$$\frac{e_1 \in T_1 \to T_2 \qquad e_2 \in T_1}{e_1(e_2) \in T_2} \qquad \frac{e \in \boldsymbol{N}}{e.a \in \tau(a)}$$

Figure 5. Saiga's core type inference rules.

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \longrightarrow \sigma' \vdash \texttt{IF}\ e_1'\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3} \text{ (CondLeft)}$$

$$\frac{}{\sigma \vdash \texttt{IF}\ \mathit{true}\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2 \longrightarrow \sigma \vdash e_1} \text{ (CondTrue)}$$

$$\frac{}{\sigma \vdash \texttt{IF}\ \mathit{false}\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2 \longrightarrow \sigma \vdash e_2} \text{ (CondFalse)}$$

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash e_1(e_2) \longrightarrow \sigma' \vdash e_1'(e_2)} \text{ (AppLeft)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash v(e) \longrightarrow \sigma' \vdash v(e')} \text{ (AppRight)}$$

$$\frac{}{\sigma \vdash v_1(v_2) \longrightarrow \sigma \vdash v_1\ v_2} \text{ (AppApp)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash e.a \longrightarrow \sigma' \vdash e'.a} \text{ (AttrLeft)}$$

$$\frac{\sigma(n, a) = e}{\sigma \vdash n.a \longrightarrow \sigma \vdash e} \text{ (AttrApp)}$$

Figure 6. Saiga's basic small-step operational semantics.

Saiga's core type inference rules are shown in Figure 5 and are standard except for the typing of attribute value expressions that simply uses τ to obtain the declared type of the attribute. To obtain a complete type inference definition, the rules in Figure 5 must be augmented by rules for other basic types and operations on those types. We leave these rules unspecified since they don't influence attribute evaluation. We also refrain from naming the type rules since we do not refer to them explicitly in the following.

In the following we assume that an expression e submitted for evaluation is well-typed in the sense that the type inference rules are able to infer a unique type for e.

3.2 Basic Attribute Grammars

Figure 6 shows Saiga's basic small-step semantic rules that define the judgement $\sigma \vdash e \longrightarrow \sigma' \vdash e'$ which means that when expression e is evaluated in context σ it steps to the

expression e' and the new context σ'. (At this stage, σ' will always be the same as σ. We make use of context update to model caching semantics in Section 3.4.)

The first six rules are mostly standard and specify evaluation of conditional expressions and function calls. Conditional expressions have their condition stepped to a value (CondLeft) and then the expression is replaced by the appropriate left or right branch expression (CondTrue or CondFalse). Applications have their function expression (AppLeft) and then argument expression (AppRight) stepped to values (thus evaluation is strict) and then the whole expression is replaced by the result of calling the function (AppApp). The latter call ($v_1\ v_2$) takes place in the underlying functional layer and can't refer to attributes, thus its evaluation mechanism is assumed.

Attribute value expressions have their node expression stepped to a node value (AttrLeft). When this value n is reached, the expression is replaced by the definition of a at n, in other words, by $\sigma(n, a)$ (AttrApp). The rule does not care what kind of expression is returned by the context, but usually an intrinsic attribute would return a value, while extrinsic attributes would usually return more complex expressions that need to be evaluated further.

It is important to note that no special mechanisms are needed to support reference attributes (i.e., attributes whose values are of node type). The rules for "normal" attributes suffice to evaluate reference attributes. Also, we do not have a guarantee that evaluation terminates, since an attribute occurrence may depend on itself. Future work will extend the semantics to detect attribution cycles and fail gracefully.

3.3 Parameterised Attributes

An attribute with a parameter can be modelled in Saiga by having an attribute that returns a function value and then applying the attribute value to the parameter. However, this approach does not lend itself to caching since function application happens outside the world of attributes. So we also model proper parameterised attributes as a core feature. See Section 5 for an extended example that uses parameterised attributes to perform name analysis.

Figure 7 shows the modified expression syntax that replaces the attribute value form $e\,.\,a$ from Figure 2 with a parameterised version $e_1\,.\,a(e_2)$. An expression of the form $e_1\,.\,a(e_2)$ passes the value of e_2 to the attribute a defined at the node given by e_1. We define only single parameter attributes for simplicity; multiple parameters can be accommodated by using tuple values. To recover non-parameterised attributes, we assume a unit type and pass the unit value *unit* to the attribute if the parameter is irrelevant. The old notation $n\,.\,a$ is then syntactic sugar for $n\,.\,a(\mathit{unit})$.

In this new scheme, the context gains an argument to specify the parameter value and we add a global function ρ

$$
\begin{array}{rcll}
e & ::= & v & \text{value} \\
 & | & \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 & \text{conditional} \\
 & | & e_1(e_2) & \text{function apply} \\
 & | & e_1\,.\,a(e_2) & \text{attribute value}
\end{array}
$$

Figure 7. Expression abstract syntax using parameterised attributes.

$$\frac{e_1 \in \boldsymbol{N} \qquad e_2 \in \rho(a)}{e_1\,.\,a(e_2) \in \tau(a)}$$

$$\frac{\sigma \vdash e_1 \longrightarrow \sigma' \vdash e_1'}{\sigma \vdash e_1\,.\,a(e_2) \longrightarrow \sigma' \vdash e_1'\,.\,a(e_2)}\ \text{(AttrLeft)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash n\,.\,a(e) \longrightarrow \sigma' \vdash n\,.\,a(e')}\ \text{(AttrRight)}$$

$$\frac{\sigma(n, a, v) = e}{\sigma \vdash n\,.\,a(v) \longrightarrow \sigma \vdash e}\ \text{(AttrApp)}$$

Figure 8. The new rules to type and evaluate parameterised attribute value expressions.

that gives the parameter types for attributes. Thus, we have

$$\rho \in \boldsymbol{A} \to \boldsymbol{T}$$

and the context now has the following dependent type:

$$\sigma \in \boldsymbol{N} \times (a : \boldsymbol{A}) \times \rho(a) \to \boldsymbol{E}$$

A correctly-typed context now obeys

$$\tau(a) = t \implies \forall n \in \boldsymbol{N}, v \in \rho(a).\sigma(n, a, v) \in t$$

Figure 8 gives the new versions of the type inference and step rules for attribute value expressions. Typing an attribute value expression now also checks the type of the parameter. Evaluation now steps the node expression (AttrLeft), then the parameter expression (AttrRight). When both are values, the context is used to obtain the attribute definition (AttrApp).

3.4 Attribute Caching

Many attribute grammar systems support attribute caching, so that attribute occurrences do not need to be evaluated more than once. Up until now, the context in our formalism has been passed unchanged from step to step. Now we update the context whenever we determine the value of an attribute occurrence. The effect is that subsequent uses of the same attribute occurrence will receive that value instead of the expression from which that value was derived.

We introduce a new syntactic construct, shown in Figure 9, that remembers the "source" of an expression, so that when we determine the final value of the expression we know under which attribute occurrence to cache it. An expression of the form $n\,.\,a(v) := e$ means that we are evaluating e and

$$\begin{array}{rcll} e & ::= & \ldots & \text{Figure 7} \\ & | & n.a(v) := e & \text{caching} \end{array}$$

Figure 9. Extension to the expression abstract syntax for attribute caching.

$$\frac{e \in \tau(a) \qquad v \in \rho(a)}{n.a(v) := e \in \tau(a)}$$

$$\frac{\sigma(n,a,v) = e}{\sigma \vdash n.a(v) \longrightarrow \sigma \vdash n.a(v) := e} \text{ (AttrApp)}$$

$$\frac{\sigma \vdash e \longrightarrow \sigma' \vdash e'}{\sigma \vdash n.a(v) := e \longrightarrow \sigma' \vdash n.a(v) := e'} \text{ (CacheLeft)}$$

$$\frac{\sigma' = \sigma \oplus \{(n,a,v_1) \mapsto v_2\}}{\sigma \vdash n.a(v_1) := v_2 \longrightarrow \sigma' \vdash v_2} \text{ (CacheWrite)}$$

Figure 10. New operational and type inference rules for attribute caching.

when we have the final value it should be cached as the value of the attribute occurrence $n.a(v)$.

The new and modified type inference and semantic rules that achieve caching are shown in Figure 10. A cache expression of the form $n.a(v) := e$ is well-typed at $\tau(a)$ if e is and $v \in \rho(a)$. The new AttrApp rule shows that now when an attribute definition is extracted from the context function, we get a cache expression containing the definition, not just the definition itself. The rest of the cache expression remembers the attribute occurrence. The expression in a cache construct is stepped as usual until a value is obtained (CacheLeft). When a value is obtained, the context is updated to record the final value of the occurrence (CacheWrite). (We use $\oplus$ to denote a point update of a context function.)

CacheWrite is the only semantic rule that modifies the context function, and it ensures that a context function will return a non-value expression only once for any given set of parameters, thereafter returning the value that the first expression was resolved into. Since the value v_1 is used as part of the context argument, parameterised attributes are cached per parameter which is the expected behaviour in systems such as JastAdd [2] and Kiama [14]. If $v_1 = \mathit{unit}$ then we recover the one-cache-per-attribute behaviour of caching for non-parameterised attributes.

This expression of the caching mechanism demonstrates the usefulness of the context function mechanism. Our semantic rules are not concerned with how the context function remembers what has been cached and what has not; they just retrieve expressions, and sometimes tell the context to override some expressions for the future.

3.5 Mechanisation

We mechanised the Saiga language and semantics in Coq using standard techniques [11]. The expression grammar is defined as a dependent inductive type. The small-step semantics is defined as an inductive relation between pairs of context functions and typed expressions. Instead of defining type inference for expressions as a relation, we incorporate types as expression parameters, so they are typed by construction. In other words, $\boldsymbol{E}$ is parameterised by some $t \in \boldsymbol{T}$. For example, *true* is an expression of type $\boldsymbol{E}(\mathit{Bool})$. Using dependent types for expressions means that the context can also be typed more precisely as:

$$\sigma \in \boldsymbol{N} \times (a : \boldsymbol{A}) \times \rho(a) \to \boldsymbol{E}(\tau(a))$$

and it is therefore not possible for the context to return an expression that is inappropriate for the attribute a. Similarly, when values are added to the context they must be of the appropriate type.

The mechanisation contains cached and parameterised attributes as described above. Unlike the semantics shown in this paper, we also include options to enable or disable caching on a global or per-attribute basis. We found that proving lemmas about particular programs or the system itself is easier with caching disabled, and we wanted to have per-attribute caching control so we can analyse the performance implications of caching in a granular way.

4 Meta-theoretic Properties

One of our key goals in developing a formalisation for attribute grammar evaluation was to be able to prove properties about programs of that system and also prove properties about the system itself. Section 5 considers some specific attribute grammars. In this section we discuss important general properties that have been mechanically proven and some more interesting properties that have not been. For the latter, we provide a high-level outline of how they might be proven.

4.1 General Properties

First, general properties that have all been mechanised in our Coq development. Determinism states that any context, expression pair steps to at most one other context, expression pair.

Theorem 4.1 (Determinism)**.**

$$\forall \sigma, \sigma_1', \sigma_2', e, e_1', e_2'. \sigma \vdash e \longrightarrow \sigma_1' \vdash e_1' \wedge \sigma \vdash e \longrightarrow \sigma_2' \vdash e_2'$$
$$\implies e_1' = e_2' \wedge \sigma_1' = \sigma_2'$$

It is not difficult to see that both of the uncached and cached versions of the semantics are syntax-directed, so this proof is straightforward.

Strong progress states that any well-formed expression is either a value or evaluation can take a step from it.

Theorem 4.2 (Strong Progress)**.**

$$\forall \sigma, e.\, value(e) \vee \exists \sigma', e'.\, \sigma \vdash e \longrightarrow \sigma' \vdash e'$$

Similarly to determinism, we can observe that well-formed non-values can each make a step via a semantic rule, so again the proof is straightforward.

Finally, type preservation states that stepping an expression will not change its type.

Theorem 4.3 (Type Preservation)**.**

$$\forall \sigma, \sigma', e, e'.\, e \in t \wedge \sigma \vdash e \longrightarrow \sigma' \vdash e' \implies e' \in t$$

As noted earlier, in our Coq implementation we define expressions to be dependent upon their type. We also define the 'step' relation to be dependently typed, such that it is impossible to create a step rule that breaks type preservation. This means that our mechanisation gets type preservation for free, without needing to be manually proven.

On paper, most of the proof for this property follows from induction. The AppApp rule depends on v_1 and v_2 having corresponding types $t_1 \to t_2$ and t_1. The type inference rules for application expressions (Figure 4) say the expression (left-hand side of the step) has type t_2, and the mathematical result of the function application (right-hand side of the step) also yields a t_2.

The AttrApp rule relies on the context being well-typed, as defined in Sections 3.1 and 3.3. The type inference rule for an attribution expression (Figure 4) gives an attribution expression a type extracted from the attribute: $\tau(a)$. The rule for type-correctness of a context also gives that an expression extracted from the context has the type $\tau(a)$.

4.2 Cache Irrelevance

Cache irrelevance is much more difficult to prove than the previous properties. This property states that if we begin with any well-formed context and expression, and this expression eventually evaluates to a value, the value obtained with caching is the same as that obtained without caching. This property can be stated as follows, where $\xrightarrow{cached}{}^*$ refers to a series of steps using all extensions, and $\xrightarrow{uncached}{}^*$ refers to a series of steps without the caching extension:

Theorem 4.4 (Cache Irrelevance)**.**

$$\begin{aligned} &\forall \sigma, e, v. \\ &\quad \exists \sigma_1'.\sigma \vdash e \xrightarrow{cached}{}^* \sigma_1' \vdash v \\ \iff &\exists \sigma_2'.\sigma \vdash e \xrightarrow{uncached}{}^* \sigma_2' \vdash v \end{aligned}$$

Since we are only considering expressions that eventually terminate, we can rule out attributes that lead to circular dependencies and therefore split the cases into a) expressions that don't involve attributes at all, and b) expressions that do refer to attributes but will eventually remove all attribute value sub-expressions. In case (a), evaluation can't be affected by caching since there are no attributes, so the value produced must be the same.

In case (b), we apply structural induction and assume that all sub-expressions evaluate to the same value regardless of caching. Therefore, we know that the value calculated for an attribute without caching must be the same as the value calculated the first time with caching, since apart from the sub-expression attributes the only evaluation is in the functional layer which is not affected by attributes. Finally, we observe that the value obtained from the cache by subsequent evaluations of an attribute with caching must be the same as the first value obtained, since the CacheWrite rule updates the context with that value.

Cache irrelevance holds in our testing of the name analysis attribute grammars described in Section 5. A mechanisation of a proof of the general theorem is work in progress.

4.3 Cache Step Reduction

Given that evaluating an expression will not have a different result under caching, we would also like to know that evaluation will perform no more "work" with caching than without. We define "work" to mean attribute evaluations, computation in the functional layer, etc, but exclude the CacheWrite rule which is just concerned with updating the cache (Figure 10). Note that all other modified rules in the cached version, either have a direct analogue in the uncached version (AttrApp), or step "in time" with their underlying expression (CacheLeft).

In the formal statement of the theorem, we annotate the steps with their step counts.

Theorem 4.5 (Cache Step Reduction)**.**

$$\begin{aligned} &\forall \sigma, \sigma_1', \sigma_2', e, v. \\ &\sigma \vdash e \xrightarrow{cached}{}^{n_1} \sigma_1' \vdash v \wedge \sigma \vdash e \xrightarrow{uncached}{}^{n_2} \sigma_2' \vdash v \\ &\implies n_1 \leq n_2 \end{aligned}$$

We know that the context is only ever updated in the CacheWrite rule and this update involves overriding one particular output expression with a value. Since cache irrelevance is established, we know that this value is the same value that the overridden expression would evaluate to. Now consider what was overridden. If a value was overridden with the same value, then the context has not in fact changed. In this instance, the cache update has had no effect on any evaluation that follows, so the number of steps to a value will remain unchanged. If a more complex expression has been overridden with a value, then any evaluation that relies on this particular output will receive a value instead of an expression that would have evaluated to this value. Here we have saved some steps, and so the cached evaluation will complete in fewer steps than the uncached evaluation.

As for cache irrelevance, cache step reduction holds in our testing of the name analysis attribute grammars described in Section 5. A mechanisation of a proof of the general theorem is work in progress.

```
class A {
    int y;
    AA a;
    y = a.x;
    class AA {
        int x;
    }
    class BB extends AA {
        BB b;
        b.y = b.x;
    }
}
```

Figure 11. A PicoJava program.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Item}^{+} \\
\mathit{Item} & ::= & \mathit{Decl} \mid \mathit{Stmt} \\
\mathit{Decl} & ::= & \mathit{VarDecl} \mid \mathit{ClassDecl} \\
\mathit{VarDecl} & ::= & \mathit{Access}\ \mathit{Name} \\
\mathit{ClassDecl} & ::= & \mathit{Name}\ \mathit{Use}?\ \mathit{Block} \\
\mathit{Block} & ::= & \mathit{Item}^{*} \\
\mathit{Stmt} & ::= & \mathit{Access}\ \mathit{Exp} \\
\mathit{Exp} & ::= & \mathit{Access} \mid \mathit{BoolLit} \\
\mathit{Access} & ::= & \mathit{Use} \mid \mathit{Dot} \\
\mathit{Use} & ::= & \mathit{Name} \\
\mathit{Dot} & ::= & \mathit{Access}\ \mathit{Use} \\
\mathit{BoolLit} & ::= & \mathit{true} \mid \mathit{false}
\end{array}
$$

Figure 12. Abstract syntax of the PicoJava language.

An alternative caching result that we do not consider here could count all steps. It would not be possible to prove a reduction in steps when caching is used since the steps that maintain the cache would now be counted. But it would be interesting to add a cost model and then reason about the trade-offs between re-evaluation of attributes and the overhead of caching. We leave this result for future work.

5 Name Analysis for PicoJava

PicoJava is a cut-down version of Java which was first used as an example for the JastAdd system[2]. The language is designed to pose a realistic name analysis challenge (including associated type analysis) without the distraction of orthogonal features. Figure 11 shows a typical program from the JastAdd PicoJava specification test suite and Figure 12 shows the version of PicoJava's abstract syntax that we will use. Our version differs slightly from the JastAdd version to simplify the presentation. It restricts different constructs as follows:

- declarations: variables and classes,
- statements: assignments,
- expressions: accesses and Boolean literals, and
- types: Boolean predefined type.

For the purposes of name analysis, the key PicoJava constructs are programs and class declarations that each define a scope for their constituent items. Defining occurrences of names occur in variable declarations (where the *Access* specifies the variable's type), and in class declarations (where the optional *Use* specifies a superclass and the *Item* list is the class body). Applied occurrences of names are *Use* constructs, either directly (unqualified) or on the right-hand side of a *Dot* construct (qualified).

We assume that abstract syntax trees have fields that are named after their non-terminals. For example, if n is a *Stmt* defined by $\mathit{Stmt} ::= \mathit{Access}\ \mathit{Exp}$ then its two children are $n.\mathit{access}$ and $n.\mathit{exp}$.

5.1 Commonalities

Since the two programs we are comparing are solving the same problem, there are some parts of the specification are similar enough to be shared.

UnknownDecl is a unique declaration that is used when name analysis cannot determine a relevant defining occurrence of a general name. Similarly, *UnknownClass* is a unique class declaration for when a class is sought but can't be found (e.g., a superclass that is not defined). *UnknownClass* is assumed to have no superclass and an empty block so it can be safely searched and no identifiers will be found. Using this approach simplifies the attribute equations since we don't need to explicitly handle the special case when a class is not defined.

The function $\mathit{finddecl}(s, l)$ searches through an item list l looking for a declaration of the name s. If such a declaration is found, it is returned, otherwise *UnknownDecl* is returned.

Figure 13 shows the definitions of *superclass* and *type*, two attributes that are used by both of the name analysis approaches. *superclass* gives the *ClassDecl* that represents the superclass of a given class declaration, if there is one, otherwise it gives the special value *UnknownClass*. Similarly, *type* returns the *ClassDecl* that represents the type of a declaration. When applied to a class declaration it returns that declaration. When applied to a variable declaration it returns the class of that variable. If neither of these situations apply or if the type of a variable is not a class, *type* returns *UnknownClass*.

5.2 Notations

We use some short-hand notations in the following definitions to simplify the presentation. `IF OK` e_1 `ELSE` e_2 is equivalent to

$$\texttt{IF}\ \mathit{notequal}(e_1)(\mathit{UnknownDecl})\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2$$

In other words, if e_1 is not unknown it evaluates to e_1, otherwise to e_2.

[2] `http://jastadd.cs.lth.se/examples/PicoJava`

$$
\begin{array}{lllll}
\sigma(n, \mathit{superclass}) & n \text{ is a } \mathit{ClassDecl} \text{ with a } \mathit{Use} & = & \texttt{IF } n.\mathit{use}.\mathit{decl} \texttt{ IS A } \mathit{ClassDecl} & (1) \\
 & & & \quad \texttt{THEN } n.\mathit{use}.\mathit{decl} & \\
 & & & \quad \texttt{ELSE } \mathit{UnknownClass} & \\
 & \text{otherwise} & = & \mathit{UnknownClass} & (2) \\
 & & & & \\
\sigma(n, \mathit{type}) & n \text{ is a } \mathit{ClassDecl} & = & n & (3) \\
 & n \text{ is a } \mathit{VarDecl} & = & \texttt{IF } n.\mathit{access}.\mathit{decl} \texttt{ IS A } \mathit{ClassDecl} & (4) \\
 & & & \quad \texttt{THEN } n.\mathit{access}.\mathit{decl} & \\
 & & & \quad \texttt{ELSE } \mathit{UnknownClass} & \\
 & \text{otherwise} & = & \mathit{UnknownClass} & (5)
\end{array}
$$

Figure 13. Equations for attributes that are shared by the two name analysis methods.

`IF` e_1 `AND` e_2 `THEN` e_3 `ELSE` e_4 is equivalent to

`IF` e_1 `THEN IF` e_2 `THEN` e_3 `ELSE` e_4 `ELSE` e_4

In other words, an `AND` condition turns into nested conditional expressions.

`IF` e `IS A` t, where t is a non-terminal name, is equivalent to

$$\mathit{equal}(e.\mathit{kind})(t)$$

where *kind* is an intrinsic attribute of each node that tags it with its non-terminal type.

5.3 Environment Method

Figure 14 gives the attribute definitions for the environment approach to performing PicoJava name analysis. *decl* is only supposed to be called on *Use* nodes, but if we ask for a *Dot*'s *decl*, the *decl* of its *Use* is returned (E_1). If the node in question is on the "right-hand side" of a *Dot* (i.e., the *Use* is qualified) then *decl*'s search begins in the class of the access on the left-hand side (E_2). Otherwise, the current environment is searched (E_3).

The environment method's name analysis hinges upon the *env* and *cenv* attributes which compile ordered lists of declaration nodes that are in scope for the target node. *env* finds declarations from the lexical scope, and also includes declarations from *cenv*, which finds declarations from the class inheritance chain. *predefs* is also appended to the declaration list, so that predefined declarations can be searched if an appropriate declaration is not found in the syntax tree (E_5). *finddecl* then searches the compiled list for the appropriate declaration, returning *UnknownDecl* upon failure.

5.4 Lookup Method

Figure 15 gives the attribute definitions for the lookup approach to performing PicoJava name analysis. *decl* is much the same as the *decl* for the environment method (Section 5.3) except that the case for *Use*s on the right-hand side of a *Dot* is handled in the *lookup* attribute.

While the environment method's strategy is to collect all appropriate declarations and bring them down to the target node, the lookup method searches upward for the appropriate declaration, using the parameter of the attribute to remember the name it is looking for. The central attribute is *lookup* that defers some of its reasoning to *locallookup* and *remotelookup*. This approach was pioneered with the development of parameterised attributes in the JastAdd system [2].

The general strategy of the *lookup* method is to move up the tree syntactically, applying appropriate searches as it goes. If a *Block* (the body of a *ClassDecl*) is reached, the *locallookup* attribute is used to search the local scope (first branch of L_5). If *locallookup* fails, *remotelookup* searches up the superclass chain (second branch of L_5). Finally, if neither of these searches finds the declaration, *lookup* moves up the tree into outer scopes (default branch of L_5). The other cases for *lookup* search the appropriate class's block for a qualified use (L_6) or the local scope by default (L_8).

locallookup is only evaluated on a *Block* or a *Program*, whose children are passed into *finddecl*. *remotelookup* is only evaluated on a *Block*, and performs a *locallookup* on this block before searching up the superclass chain if there is a superclass (L_{12}). If the appropriate declaration is not found in the tree, *locallookup* eventually searches *predefs* to see if a predefined symbol matches the lookup name (L_9).

5.5 Method Equivalence

The motivator for implementing name analysis in two different ways was to compare them in a formal and structured way. The main thing we want to know is that both methods will produce the same result when determining the declaration for the same node in a tree. In Theorem 5.1, we use decl_l and decl_e to refer to the *decl* attribute from Figures 15 and 14 respectively. Similarly we use σ_l and σ_e to represent contexts which have the same intrinsic attributes but contain attribute definitions from Figures 15 and 14 respectively.

Theorem 5.1 (Lookup vs Environment Equivalence)**.**

$$
\begin{aligned}
&\forall \sigma_l, \sigma_e, e, v. \\
&\qquad \exists \sigma_1'. \sigma_l \vdash n.\mathit{decl}_l \longrightarrow^* \sigma_1' \vdash v \\
&\iff \exists \sigma_2'. \sigma_e \vdash n.\mathit{decl}_e \longrightarrow^* \sigma_2' \vdash v
\end{aligned}
$$

$\sigma(n, decl)$	n is a *Dot*	=	$n.use.decl$	(E_1)
	n is a *Use* on RHS of a *Dot*	=	$finddecl(n.name, n.parent.access.decl.type.block.cenv)$	(E_2)
	n is a *Use*	=	$finddecl(n.name, n.env)$	(E_3)
	otherwise	=	*UnknownDecl*	(E_4)
$\sigma(n, env)$	n is a *Program*	=	$n.items$ ++ $predefs$	(E_5)
	n is a *Block*	=	$n.cenv$ ++ $n.parent.env$	(E_6)
	n is a *UnknownDecl*	=	{}	(E_7)
	otherwise	=	$n.parent.env$	(E_8)
$\sigma(n, cenv)$	n is a *Program*	=	{}	(E_9)
	n is a *Block*	=	IF $equal(n.parent.superclass)$(*UnknownClass*) THEN $n.items$ ELSE $n.items$ ++ $n.parent.superclass.block.cenv$	(E_{10})
	n is a *UnknownDecl*	=	{}	(E_{11})
	otherwise	=	$n.parent.cenv$	(E_{12})

Figure 14. Equations for the environment method of PicoJava name analysis.

$\sigma(n, decl)$	n is a *Dot*	=	$n.use.decl$	(L_1)
	n is a *Use*	=	$n.lookup(n.name)$	(L_2)
	otherwise	=	*UnknownDecl*	(L_3)
$\sigma(n, lookup, s)$	n is a *Program*	=	$n.locallookup(s)$	(L_4)
	n is a *Block*	=	IF OK $n.locallookup(s)$ ELSE IF OK $n.parent.superclass.block.remotelookup(s)$ ELSE $n.parent.lookup(s)$	(L_5)
	n is a *Use* on RHS of a *Dot*	=	$n.parent.access.decl.type.block.remotelookup(s)$	(L_6)
	n is a *UnknownDecl*	=	*UnknownDecl*	(L_7)
	otherwise	=	$n.parent.lookup(s)$	(L_8)
$\sigma(n, locallookup, s)$	n is a *Program*	=	IF OK $finddecl(s, n.items)$ ELSE $finddecl(s, predefs)$	(L_9)
	n is a *Block*	=	$finddecl(s, n.items)$	(L_{10})
	otherwise	=	*UnknownDecl*	(L_{11})
$\sigma(n, remotelookup, s)$	n is a *Block*	=	IF OK $n.locallookup(s)$ ELSE IF $equal(n.parent.superclass)$(*UnknownClass*) THEN *UnknownDecl* ELSE IF OK $n.parent.superclass.block.remotelookup(s)$ ELSE *UnknownDecl*	(L_{12})
	otherwise	=	*UnknownDecl*	(L_{13})

Figure 15. Equations for the lookup method of PicoJava name analysis.

The broad strokes of a proof for this theorem are easy to understand. The *superclass* and *type* attributes are shared between the two methods, as well as the *finddecl* function. As discussed in Section 5.4, the *decl* attribute follows the same logic in each method, deferring either to *lookup* or *env*, thus the focus of the proof is on these two attributes.

The intuition of the proof comes from the observation that *lookup* applies *finddecl* to items from the local syntactic scope, then to the items from each local scope up the superclass chain, then to the original location's parent scope, and so on. Conversely, *env compiles* all items (in order) from first the local scope, then all local scopes up the superclass chain, then to the original location's parent scope, and so on, and feeds this list to *finddecl*.

A proof therefore requires showing that the concatenation of all the scopes that *lookup* examines is identical to the list of

items that *env* constructs. It follows trivially that searching a set of lists in order produces the same result as searching a concatenation of these lists.

5.6 Mechanisation

Both of the described approaches have been fully mechanised in Coq, almost exactly as shown in Figures 14 and 15. We have taken the tree shown in Figure 11 and evaluated the *decl* attribute on every *Use* node to ensure that the correct result is obtained. Our tests demonstrate that the two name analysis programs are producing the same results. A mechanisation of a proof of the equivalence theorem is work in progress.

We also ran tests to confirm that caching does not affect the final result, but does affect the number of steps required to resolve a final value. Our mechanisation can produce a trace of the steps performed or a summary view of the attributes that were evaluated. Cache irrelevance and cache step reduction holds for all performed tests.

The discussion above simplifies one aspect of the name analysis specifications to keep the presentation clear. The *finddecl* function is actually implemented as an attribute in our mechanised implementation. Searching a list of nodes for the appropriate declaration involves accessing every node's *Name* attribute, but it is not possible for a value-level function to access attribute values including intrinsic attributes such as *Name*.

6 Conclusion

The Saiga language and its semantics presented here satisfy our goal of isolating the essence of attribute evaluation from the details of particular attribute grammar system implementations. We have shown how the formalisation can be used to reason about attribute evaluation, both in general via properties such as cache irrelevance, or in terms of specific attribute grammars such as in the name analysis example. A key characteristic of our approach is that it unifies the core concepts: intrinsic attributes, extrinsic attributes and tree structure; synthesised and inherited attributes; equational definitions and caching. By defining simple general mechanisms we control the complexity of the core, thereby limiting the reasoning burden. These results show that this approach is realistic, practical and can usefully form the basis of future theoretical explorations of attribute grammar evaluation.

Current work involves extending the formalism to handle dynamic detection of attribution cycles, circular attributes that evaluate to a fixed point and higher-order attributes that extend the tree structure. The latter is another way in which the evaluation context can be updated, in this case by adding new nodes with relationships to existing ones. We are also mechanising the properties for which only informal proofs have so far been completed. More speculative future work will investigate cost models for evaluation so that the trade-offs between caching and not caching can be modelled.

Acknowledgements

Thanks to Eric Van Wyk, Görel Hedin and Matthew Roberts for discussions that helped to inspire this work and influence our approach.

References

[1] Kevin Backhouse. 2002. A Functional Semantics of Attribute Grammars. In *Tools and Algorithms for the Construction and Analysis of Systems*. Number 2280 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 142–157.

[2] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd System — Modular Extensible Compiler Construction. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 14–26.

[3] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000).

[4] Thomas Johnsson. 1987. Attribute Grammars As a Functional Programming Paradigm. In *Proceedings of the Functional Programming Languages and Computer Architecture*. 154–173.

[5] Martin Jourdan. 1984. An optimal-time recursive evaluator for attribute grammars. In *Proceedings of the International Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 167. 167–178.

[6] Uwe Kastens. 1980. Ordered Attributed Grammars. *Acta Informatica* 13, 3 (March 1980), 229–256.

[7] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (June 1968), 127–145.

[8] Pedro Martins, João Paulo Fernandes, João Saraiva, Eric Van Wyk, and Anthony Sloane. 2016. Embedding attribute grammars and their extensions using functional zippers. *Science of Computer Programming* 132, Part 1 (Dec. 2016), 2–28.

[9] Oege De Moor, Kevin Backhouse, and S. Doaitse Swierstra. 2000. First-class Attribute Grammars. *Informatica* 24 (2000).

[10] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. 1999. Aspect-Oriented Compilers. In *Generative and Component-Based Software Engineering*. Springer, Berlin, Heidelberg, 121–133.

[11] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. 2017. Software foundations. *Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html* (2017).

[12] João Saraiva. 2002. Component-Based Programming for Higher-Order Attribute Grammars. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*. 268–282.

[13] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2009. Formalising and Verifying Reference Attribute Grammars in Coq. In *Programming Languages and Systems*. Number 5502 in Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[14] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. 2013. A pure embedding of attribute grammars. *Science of Computer Programming* 78, 10 (Oct. 2013), 1752–1769.

[15] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2008. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science* 203, 2 (April 2008), 103–116.