Forwarding in Attribute Grammars for Modular Language Design

Eric Van Wyk[1], Oege de Moor[1], Kevin Backhouse[1], and Paul Kwiatkowski[2]

[1] Oxford University Computing Laboratory
[2] Microsoft Corporation

Abstract. *Forwarding* is a technique for providing default attribute definitions in attribute grammars that is helpful in the modular implementation of programming languages. It complements existing techniques such as default copy rules. This paper introduces forwarding, and shows how it is but a small extension of standard higher-order attribute grammars. The usual tools for manipulating higher-order attribute grammars, including the circularity check (which tests for cyclic dependencies between attribute values), carry over without modification. The closure test (which checks that each attribute has a defining equation) needs modification, however, because the resulting higher-order attribute grammars may contain spurious attributes that are never evaluated, and indeed that need not be defined.

1 Motivation

The modular definition of programming languages is a long-standing problem, and a lot of work has been devoted to its solution in the context of attribute grammars *e.g.* [1,3,8,10,11,13,14,19,17,21,25,29,32]. Some of these proposals take inspiration from the object-oriented paradigm, advocating the use of inheritance to achieve modularisation. Others take inspiration from functional programming, by employing higher-order functions to achieve a separation of concerns. The present paper is a modest contribution towards these developments, by showing how a certain form of inheritance called *forwarding* can be achieved in higher-order attribute grammars. In our view, forwarding is the main innovative idea in the design of the Intentional Programming system [27,30]. That system, until recently under development at Microsoft, is an environment for interactive language design, similar in spirit to many of the above attribute grammar systems.

The structure of the paper is as follows. First we present a number of motivating examples that introduce forwarding, along with the idea of production-valued attributes. Next, we show how a complete grammar that was composed using forwarding can be expanded to an ordinary higher-order attribute grammar. Finally, we demonstrate that the standard circularity test for higher-order attribute grammars can be applied to such modular descriptions. The standard closure test does however need some modification, and we argue that the desired effect can be achieved through an appropriate implementation of the circularity test. That test might thus be more appropriately named the *definedness* test.

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 128–142, 2002.

1.1 A Forwarding Example: Record Invariants

Consider a programming language with record types, and the usual *with* construct for concisely referring to the field names. Assume that we wish to add a new feature, namely that of *record invariants*, which state some invariant relationship between field values. At the end of each *with* clause, it is checked that the invariant is actually satisfied. To keep the example simple, we avoid the problem of name capture by prohibiting the invariant from referring to global variables. As we discuss in our technical report [30], the more general problem is easily solved by correctly maintaining the environment. Here is an example of a program that uses record invariants:

```
let rec_type = record { f1 :: int,  f2 :: int } invariant f1 ≡ 2 * f2
    r :: rec_type
in
    with  r begin
            f1 := 4; f2 := 8
    end
```

This program fragment in the augmented language is equivalent to the following fragment in the base language:

```
let rec_type = record { f1 :: int,  f2 :: int }
    r :: rec_type
in
    with  r begin
            f1 := 4; f2 := 8;
          if ¬f1 ≡ 2 * f2 then
            error "invariant fails"
    end
```

Our challenge is to implement this extension as a small, modular addition to the base language definition. Of course this notion of language extension, where the new feature is rewritten to existing idioms, is extremely common. It is the basis of the idea that it suffices to define a small, elegant core language on which richer features are then built.

Let us call the grammar production for the new **with** construct **with'**. Essentially we would like to define its semantics through the rewrite rule

```
with' r ss
  ⇒
with r (ss ++ if ¬r.type.invariant then error "invariant fails" )
```

In particular, we do not wish to define each of the attributes for **with'** anew: that would require detailed knowledge of all semantic aspects of the base language. Note that on the right-hand side of this rewrite, we are referring to the invariant of the type of r. This is a new piece of information that has to be added to every record type. Note that the invariant is in fact represented as a syntax tree,

so here we have an example of a higher-order attribute. Defining new language features in this way, by expanding new productions to old, is called *forwarding*.

Forwarding is similar to but subtly different from syntax [33] and semantic macros [20]. Like semantic macros, forwarding does give access to semantic information, for instance the attribute $r.type.invariant$. Such semantic information is not available in syntax macros. Forwarding is different from both semantic and syntax macros in that not all attribute queries on a new production such as **with'** are forwarded to the expanded form. For example, an attribute that defines a pretty-printing of the original program would be defined for **with'** directly. If we relied on forwarding to define the pretty-printing, it would show the expanded form, which is clearly undesirable. A typical use of forwarding thus states the expansion into primitive terms as a rewrite rule, but it also defines a number of attributes whose values are specific to the original higher-level construct. This also sets attribute-grammars-with-forwarding apart from language processors that are mainly based on reflection, such as MetaML [26] or 'C [9].

Forwarding is very close to higher-order attribute grammars. As we shall see shortly, the only substantial difference is that here the "copy rules" for all relevant attributes are automatically generated. There are a number of minor differences, in particular that forwarding is commonly used in conjunction with production-valued attributes. This feature also highlights the difference between forwarding and object-oriented extensions of attribute grammars since the forwarded to construct is *dynamically* computed at attribute evaluation time instead of *statically* determined via inheritance [21] when the attribute grammar is defined. We now turn to an example to illustrate this phenomenon.

1.2 A Production-Valued Attribute Example: Operator Overloading

The aim is to overload the + operator for numeric addition and string concatenation. Furthermore, we would like to achieve this in a modular way fashion: overloading + on yet another type (such as matrices) should not require any changes to existing attribute definitions. Below we shall present three versions of the solution: the first achieves the desired modularity, the second exemplifies how production-valued attributes can be compiled away, and the final version demonstrates the reduction to ordinary higher-order attribute grammars.

Our starting point is a grammar with nonterminals *Expr* (for expressions) and *Type* (for types) that includes the following productions:

name	description	production
plus	overloaded +	*Expr ::= Expr Expr*
add	numeric addition	*Expr ::= Expr Expr*
cat	string concatenation	*Expr ::= Expr Expr*
num	numeric constant	*Expr ::= Number*
str	string constant	*Expr ::= Qstring*
id	identifiers	*Expr ::= Id*
int	type of integers	*Type* $::= \varepsilon$
string	type of strings	*Type* $::= \varepsilon$

The *Expr* nonterminal has an attribute *code* of type *String* and a higher-order attribute *type* which contains a tree derived from nonterminal *Type*. It also has an inherited attribute *environment*. Figure 1 sketches the attribute definitions that one would expect on those productions that have no direct relation to overloading of +. Note the use of *type* as a higher-order attribute. The *environment* attribute is defined implicitly by default copy. Below we shall discuss the attribute definitions for the *Type* productions, and for overloaded +.

$$
\begin{array}{l}
add : Expr ::= Expr\ \ Expr \\
\quad Expr_1.code = gen\_add\_code(Expr_2.code, Expr_3.code) \\
\quad Expr_1.type = integer \\
\\
concat : Expr ::= Expr\ \ Expr \\
\quad Expr_1.code = gen\_concat\_code(Expr_2.code, Expr_3.code) \\
\quad Expr_1.type = string \\
\\
numeric\_const : Expr ::= num \\
\quad Expr_1.code = gen\_num\_code(num) \\
\quad Expr_1.type = integer \\
\\
string\_const : Expr ::= str \\
\quad Expr_1.code = gen\_str\_code(str) \\
\quad Expr_1.type = string \\
\\
identifier : Expr ::= id \\
\quad Expr_1.code = gen\_id\_code(id) \\
\quad Expr_1.type = lookup(Expr.environment, id.lexeme)
\end{array}
$$

Fig. 1. Standard attribute definitions

Using forwarding and production-valued attributes. The productions for the types are ε-productions (empty right hand sides). We introduce a new attribute on types, called *plusProd* (short for "plus production"). The values of this attribute are tree constructors: they take two trees, and build a new tree. Furthermore these trees should be syntax trees that were derived from the *Expr* nonterminal, so the type of *plusProd* is $Expr \times Expr \rightarrow Expr$. Any production in the grammar that rewrites an expression to two further expressions could be viewed as a function of this type. For integers, *plusProd* is *add* — that is the production for numeric addition. For strings, *plusProd* is *concat*, which is the production for string concatenation. These are the formal definitions:

$$
\begin{array}{l}
integer : Type ::= \varepsilon \\
\quad Type.plusProd = add
\end{array}
\qquad\qquad
\begin{array}{l}
string : Type ::= \varepsilon \\
\quad Type.plusProd = concat
\end{array}
$$

The presence of the *plusProd* attribute and forwarding makes it rather easy to define the overloaded plus attributes. The generic plus production forwards

to a node created by the appropriate production (here *add* or *concat*) which is retrieved from an attribute on a child. This production is provided with children (the same ones as the original *plus*) and provided with inherited attributes by "inserting" it into the current tree as the third implicit child of *plus*. This is a well-known use of higher order attributes. What is new is that we are passing a production as an attribute and providing it with children. Furthermore, any synthesised attribute of the generic plus that is not explicitly defined is obtained by implicit copying from the newly created node. We have added a pretty printing attribute to this example as an example of an explicitly defined attribute. Similarly, inherited attributes are implicitly passed to the fowarding node.

$$
\begin{aligned}
&\mathit{plus} : \mathit{Expr} ::= \mathit{Expr}\ \ \mathit{Expr}\ \ \mathit{Type}\\
&\quad \mathit{Type} = \mathit{Expr}_2.\mathit{type}\\
&\quad \mathit{Expr}_1.\mathit{pp} = \mathit{Expr}_2.\mathit{pp} \mathbin{++} \text{“+”} \mathbin{++} \mathit{Expr}_3.\mathit{pp}\\
&\quad \textbf{forwardsTo}\ \mathit{Type}.\mathit{plusProd}(\mathit{Expr}_2, \mathit{Expr}_3)
\end{aligned}
$$

Note that we need not define the inherited attributes of Expr_2 and Expr_3 unless we use synthesised attributes that depend on them. For example, $\mathit{Expr}_3.\mathit{type}$ is not used, so there is no need to define $\mathit{Expr}_3.\mathit{environment}$. Unfortunately, the standard closure test requires every inherited attribute to be defined regardless. This problem and its solution are discussed further in Section 3.3.

At the outset we stated that the overloading should be achieved in a modular fashion, so that adding a new overloading is just a local change to the attribute grammar. Indeed, this goal has been achieved. All we need to add for overloading + on matrices are the following two productions:

$$
\begin{aligned}
&\mathit{matrix} : \mathit{Type} ::= \varepsilon\\
&\quad \mathit{Type}.\mathit{plusProd} = \mathit{matrix\_add}\\
&\\
&\mathit{matrix\_add} : \mathit{Expr} ::= \mathit{Expr}\ \ \mathit{Expr}\\
&\quad \mathit{Expr}_1.\mathit{type} = \mathit{matrix}\\
&\quad \mathit{Expr}_1.\mathit{code} = \mathit{gen\_matrix\_add\_code}(\mathit{Expr}_2.\mathit{code}, \mathit{Expr}_3.\mathit{code})
\end{aligned}
$$

Some readers may argue that this form of overloading is rather awkward compared to the overloading features of a modern programming language such as Haskell [16]. That is certainly true: here we merely use the example of overloaded syntax to illustrate the merits of production-valued attributes and forwarding in a nutshell.

Elimination of production-valued attributes. We now aim to show how the device of production-valued attributes can be eliminated from this example. Naturally the productions of Figure 1 remain as before. The elimination of production-valued attributes is very similar to the elimination of higher-order functions from more general programs [12].

For each production-valued attribute *attr* which can be given the value of productions $\mathit{prod}_1, \mathit{prod}_2, \ldots, \mathit{prod}_n$ create an enumerated type $\mathit{attr\_token}$ whose possible values are the tokens $\mathit{attr\_prod}_1, \mathit{attr\_prod}_2, \ldots, \mathit{attr\_prod}_n$. This is possible because there are a fixed number of productions in the grammar. We then

replace production valued attributes with attributes whose value are these new enumerated types. We replace production references in attribute definitions with the appropriate token and replace attribute references with case statements which switch on the token value to make use of the appropriate production as below.

$$
\begin{array}{l}
plus : Expr ::= Expr\ \ Expr\ \ Type \\
\quad Type = Expr_2.type \\
\quad \textbf{forwardsTo}\ case\ Type.plusProd\_token\ of \\
\qquad\qquad\qquad plusProd\_add \rightarrow add(Expr_2, Expr_3) \\
\qquad\qquad\qquad plusProd\_concat \rightarrow concat(Expr_2, Expr_3)
\end{array}
$$

$$
\begin{array}{l}
integer : Type ::= \varepsilon \\
\quad Type.plusProd\_token = plusProd\_add
\end{array}
$$

$$
\begin{array}{l}
string : Type ::= \varepsilon \\
\quad Type.plusProd\_token = plusProd\_concat
\end{array}
$$

This is still a fairly painless way to implement operator overloading. If new attributes, say for a new target language, are introduced onto the *add* and *concat* productions, we do not need to change the *plus* production, thanks to forwarding. We do lose modularity in another dimension, however: to overload matrix addition, we need to change the forwards–to clause of the *plus* production so that the *case* statement recognises a token for matrix addition.

Elimination of forwarding. We now consider how the above version of our example can be implemented without forwarding. The elimination of forwarding involves two changes. First, we need to introduce a new, explicit child of *plus* to represent the newly constructed tree that attribute queries are forwarded to. Second, all attribute definitions for *plus* have to be made explicit. The inherited attributes of the newly constructed tree are the inherited attributes of the generic plus. The synthesised attributes of the generic plus are the same as the synthesised attributes of the newly constructed tree. In summary, the *plus* production is transformed into the following:

$$
\begin{array}{l}
plus : Expr ::= Expr\ \ Expr\ \ Type\ \ Expr \\
\quad Type = Expr_2.type \\
\quad Expr_4 = case\ \ Type.plusProd\_token\ \ of \\
\qquad\qquad\quad plusProd\_add \rightarrow add(Expr_2, Expr_3) \\
\qquad\qquad\quad plusProd\_concat \rightarrow concat(Expr_2, Expr_3) \\
\quad Expr_1.code = Expr_4.code \\
\quad Expr_1.type = Expr_4.type
\end{array}
$$

No other productions need to be altered. It goes without saying that most of the modularity has been lost at this stage. In particular, the introduction of a new target language would necessitate a new attribute equation, and the introduction of a new overloading would require a new clause in the case expression.

2 Attribute Grammars with Forwarding

2.1 Definition of Attribute Grammars with Forwarding

An attribute grammar with forwarding is defined as a tuple $\langle G, A, S\rangle$ where G is a context free grammar, A specifies attributes for nonterminals in G and S defines the attribute defining semantic functions for each production in G.

A *context free grammar* G is defined as a tuple $\langle N, T, P, S\rangle$ where N is a finite set of nonterminal symbols, T is a finite set of terminal symbols, S is a nonterminal in N ($S \in N$) called the *start symbol* and $P \subset N \times (N \cup T)^*$ is finite set of productions. Each $p \in P$ has the form $X_0^p ::= X_1^p\ X_2^p\ \ldots\ X_{m_p}^p\ X_{f_p}^p$ where $X_0^p \in N$ is the left hand side of the production p, $X_i^p \in N \cup T, 1 \leq i \leq m_p$ are the standard terminals and nonterminals on the right hand side of the production p, and $X_{f_p}^p, f_p = m_p + 1$, is a distinguished optional nonterminal called the *forwards-to* nonterminal. The left hand side nonterminal X_0^p is the same nonterminal as the forwards-to nonterminal $X_{f_p}^p$, if it exists. If there is no forwarding nonterminal for p then $f_p = m_p$.

Each nonterminal is attributed with semantic values called attributes. For a nonterminal $X \in N$, $A(X)$ is the set of attributes which are assigned values for X. This set is partitioned into synthesised, $A_s(X)$, and inherited, $A_i(X)$, attributes. The set of all attributes is $A = \bigcup_{X \in N} A(X)$. The *type* of an attribute $a \in A$ is specified by $A_t(a)$ and indicates the possible values that can be assigned to occurrences of a. A set of *base* types T_b is left undefined but typically includes integers, strings, etc. In traditional attribute grammars defined by Knuth [18], $A_t(a) \in T_b$. In higher order attribute grammars [29,32,31] attributes can also take on the value of syntax trees whose type is the terminal or nonterminal symbol at the root of the tree. Thus $A_t(a) \in T_b \cup N \cup T$.

In higher order attribute grammars, some of the nonterminals on the right hand side of the production are classified as *nonterminal attributes*. The abstract syntax trees rooted on these nonterminals are not created by parsing a source text, as the standard nonterminals are, but are generated by semantic rules associated with the production. We will require that all nonterminals classified as nonterminal attributes are to the right of all standard nonterminals on the right hand side of the production. Thus, we can define nta_p as the index of the first nonterminal attribute of p such that for every $i \geq nta_p, X_i^p$ is a nonterminal attribute and for every $i < nta_p, X_i^p$ is a standard nonterminal. In particular, the *forwards-to* nonterminal is also a *nonterminal attribute*. If p has no nonterminal attributes then $nta_p > m_p$. The *signature* of a production p, denoted $\sigma(p)$, is $X_1^p \times X_2^p \times \ldots \times X_{nta_p-1}^p \rightarrow X_0^p$ - the right hand side nonterminals whose trees are not computed by semantic rules on p and the left hand side nonterminal. The set of all signatures for all productions is $\Sigma(P) = \bigcup_{p \in P} \sigma(p)$.

To assign values for nonterminal attributes it is often convenient to use *production-valued attributes*. The productions passed via these attributes are applied to the appropriate trees to produce the tree to be assigned to the nonterminal attribute. We thus extend the possible types of attributes to allow for production valued attributes so that $A_t(a) \in T_b \cup N \cup T \cup \Sigma(P)$.

For each production $p = X_0^p ::= X_1^p\ X_2^p\ \ldots\ X_{m_p}^p\ X_{f_p}^p \in P$, we have a set of semantic rules $S(p)$ for computing values of synthesised attributes for X_0^p, $a \in A_s(X_0^p)$, inherited attributes of $X_i^p, 1 \leq i \leq f_p$, $a \in A_i(X_i^p)$ and nonterminal attributes $X_i^p, nta_p \leq i \leq f_p$. The set $A_s^e(p), p \in P$ is the set of synthesised attributes defined explicitly by a semantic rule in $S(p)$ for nonterminal X_0^p and $A_i^e(p, X_j^p), p \in P, j \geq 1$ is the inherited attributes defined explicitly by a semantic rule in $S(p)$ for X_j^p. In standard (higher order) attribute grammars, it is required that $S(p)$ contains a semantic rule for each attribute $a \in A_s(X_0^p)$ and each attribute $a \in A_i(X_i^p), \leq i \leq m_p$ and each nonterminal attribute $X_i^p, nta_p \leq i \leq f_p$. That is, $A_s(X_0^p) = A_s^e(p)$ and $A_i(X_i^p) = A_i^e(p, X_i^p), \forall i.1 \leq i \leq f_p$. For productions using forwarding, we only require that all nonterminal attributes are explicitly defined by rules in $S(p)$. As we will see, synthesised attributes $a \in A_s(X_0^p)$ which are not explicitly defined receive as their value the value of $X_{f_p}^p.a$ and inherited attributes $a \in A_s(X_i^p), 1 \leq i \leq f_p$ which are not explicitly defined are not needed in the calculation of the synthesised attributes on X_0^p. A *definedness* test verifying that this condition holds is discussed in Section 4.

2.2 Attribute Evaluation

Attribute grammars with forwarding can be evaluated directly, as described here, or embedded into standard higher order attribute grammars and evaluated in that framework by traditional means as described in Section 3.

Attribute evaluation proceeds as it normally does for higher order attribute grammars with the exception of synthesised attributes not explicitly defined by a production p ($a \in A_s(X_0^p) \setminus A_s^e(p), p \in P$) and inherited attributes for the forwards-to nonterminal which are not explicitly defined ($a \in A_i(X_{f_p}^p) \setminus A_i^e(p, X_{f_p}^p), p \in P$). For synthesised attribute occurrences a on nonterminals X_0^p defined by production p such that $a \in A_s(X_0^p) \setminus A_s^e(p)$, that is, those for which there is no defining semantic rule in $S(p)$, we will use the value $X_{f_p}^p.a$. That is, if X_0^p is queried for its a attribute value it will return $X_0^p.a$ if there is a semantic rule in $S(p)$ defining a ($a \in A_s^e(p)$), otherwise it returns $X_{f_p}^p.a$. For inherited attributes a not explicitly defined for $X_{f_p}^p$ by p, $a \in A_i(X_{f_p}^p) \setminus A_i^e(p, X_{f_p}^p)$ we copy the values from X_0^p.

The direct evaluation described here is particularly easy to implement by encoding the attribute grammar as a lazy functional program [15,1] and forms the basis of our prototype Intentional Programming system [30].

3 Reduction to Higher Order Attribute Grammars

Forwarding enables the decomposition of an attribute grammar into separate aspects, which are fragments that define a group of related attributes [7,6]. Once all the aspects are known, and a complete grammar is woven from the pieces, forwarding can be eliminated. That is important both for the implementation and analysis of an attribute grammar. Much earlier work on efficient evaluation

can be used directly, and the tools for analysing attribute grammars need only be modified to trace potential errors to the source that used forwarding. This section is divided into three parts: first we show how production-valued attributes are eliminated, and then we demonstrate how forwarding itself can be transformed away. Finally we discuss how well the standard closure test, circularity test and attribute evaluation strategies work with the reduced grammar. Some readers may find it helpful to study the formal descriptions below alongside the concrete example in Section 1.

3.1 Elimination of Production-Valued Attributes

As mentioned before, our technique for eliminating production-valued attributes is very similar to the *de-functionalisation* of higher-order programs. That transformation was first proposed and studied by Reynolds [24], see also [5]. It is a whole-program transformation where function types are replaced by an enumeration of the function abstractions in the program.

Here, we introduce an enumeration type for all production names. Next, we replace each production-valued attribute *attr* with an enumeration valued attribute $attr_{pn}$ generated from the names of the intended productions. Furthermore replace each reference to a production-valued attribute and its application to trees $t_1,\ t_2, \ldots, t_n$

$$X_j^p.attr(t_1, t_2, ..., t_n)$$

by the expression

$$\begin{array}{l} case\ X_j^p.attr_{pn}\ of \\ \quad attr\_p_1 \rightarrow p_1(t_1, t_2, ..., t_n) \\ \quad attr\_p_2 \rightarrow p_2(t_1, t_2, ..., t_n) \\ \quad ... \\ \quad attr\_p_m \rightarrow p_m(t_1, t_2, ..., t_n) \end{array}$$

such that $attr\_p_i, 1 \leq i \leq m$ is the enumeration token value for production p_i such that $\sigma(p_i) = A_t(a), p_i \in P, 1 \leq i \leq m$. As the defunctionalisation transformation is well-known, and this is a particularly simple instance, we confine its exposition to this brief sketch.

3.2 Elimination of Forwarding

Our starting point is an attribute grammar with forwarding, as defined in Section 2.1. The forwarding is eliminated in two steps, with a third optional step that is necessitated for the result to be acceptable in many attribute grammar systems.

1. *Add semantic rules to explicitly copy synthesised attribute values from the forwarding nonterminals to the left hand side nonterminals.* For each forwarding production $p \in P$ and for each attribute $a \in A_s(X_0^p) \setminus A_s^e(p)$ add the following semantic rule:

$$X_0^p a. = X_{f_p}^p.a$$

That is, for each synthesised attribute a that is declared to annotate the left hand side of p ($a \in A_s(X_0^p)$) but is not one of the attributes explicitly defined by p ($a \notin A_s^e(p)$), add the above semantic rule to p.

2. *Add semantic rules to explicitly copy inherited attribute values from the left hand side nonterminals to the forwarding nonterminals.* For each forwarding production $p \in P$ and for each attribute $a \in A_i(X_{f_p}^p) \setminus A_i^e(p, X_{f_p}^p)$ add the following semantic rule:

$$X_f^p.a = X_0^p.a$$

That is, for each inherited attribute a that is declared to annotate the forwards-to nonterminal $X_{f_p}^p$ of p ($a \in A_i(X_{f_p}^p)$) but is not one of the attributes explicitly defined by p ($a \notin A_i^e(p, X_{f_p}^p)$), add the above semantic reul to p.

3. *Add semantic rules for undefined inherited attributes.* This step is optional and is only necessary to force the reduced higher order attribute grammar definition to pass the standard closure tests. It does not affect the evaluation of the attribute grammar. For each forwarding production $p \in P$ and for each attribute $a \in A_i(X_j^p) \setminus A_i^e(p, X_j^p), 1 \leq j \leq m_p$ add the following semantic rule:

$$X_j^p.a = \alpha_{A_t(a)}$$

where $\alpha_{A_t(a)}$ is any value of type $A_t(a)$. That is, for each inherited attribute a that is declared to annotate the nonterminal $X_j^p, 1 \leq j \leq m_p$ of p ($a \in A_i(X_j^p)$) but is not one of the attributes explicitly defined by p ($a \notin A_i^e(p, X_j^p)$), add the above semantic rule to p.

Similar mechanisms for the automatic generation of copy rules first came to our attention when studying the micro attribute grammar system produced by Swierstra and his colleagues at Utrecht [28]. That system does however not provide forwarding.

3.3 Closure, Circularity and Attribute Evaluation

Once the attribute grammar with forwarding has been reduced we can apply the standard closure and circularity tests and use existing mechanisms for attribute evaluation. We have, in fact, developed a simple prototype which uses the process described above to reduce a grammar with forwarding to a standard higher order attribute grammar written in SSL, the attribute grammar definition language of the Synthesizer Generator [23]. This allows us to use this tool's analysis tests and attribute evaluation implementation.

Attribute grammars are typically checked for definedness in two phases. The first phase, known as the *closure test*, checks that no semantic rules are missing. For example, if somewhere in the grammar the synthesised attribute *code* from a subtree of type *Stmt* is used, then every production with *Stmt* on its left hand side must provide a semantic function for *code*. The second phase, known as the *circularity test*, checks whether there is an input tree on which the attributes are circularly defined. We can safely apply the circularity test to the reduced

grammar since circularities in the original grammar will also be detected as circularities in the reduced grammar. However, there are problems with the closure test. Although we can force the reduced grammar to pass the standard closure test (step 3 above), it then fails to detect genuine missing definitions. Also, the Synthesizer Generator's strict evaluation strategy on the reduced grammar will cause the unnecessary evaluation of unused attributes.[1] The root of both problems is that a production can define values for the left hand side nonterminal either explicitly or implicitly via forwarding.

Consider using forwarding to define a *for* loop in terms of the expected *while* loop as defined below. Here we have used quoting and implicit anti-quoting functions in order to specify the forwards-to construct using its concrete syntax instead of the abstract syntax tree constructors as we've done before.

$$
\begin{aligned}
&\mathit{for} : \mathit{Stmt} ::= \mathit{id}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Stmt}\\
&\quad \mathit{Stmt}_1.\mathit{pp} = \mathit{gen\_for\_pp}(\mathit{id}_1, \mathit{Expr}_1.\mathit{pp}, \mathit{Expr}_2.\mathit{pp}, \mathit{Stmt}_2.\mathit{pp})\\
&\quad \textbf{forwardsTo}\ \text{parse}\ \text{“}\mathit{id}_1 := \mathit{Expr}_1\ \text{; while (}\ \mathit{id}_1 \leq \mathit{Expr}_2\text{) do}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Stmt}_2\ ;\ \mathit{id}_1 := \mathit{id}_1 + 1\ \text{endwhile”}
\end{aligned}
$$

Except for the pretty print attribute, the semantics of *for* are determined entirely by forwarding. The efficiency problem can be seen by considering a strategy which evaluates all attribute occurrences in the tree. Such a strategy would unnecessarily compute the *code* attribute for the nodes in the child trees of *for* and the pretty print attribute for the nodes in the forwards-to tree. In contrast, demand driven evaluation would only evaluate those attribute definition functions which are necessary.

The problems with the closure test are more subtle. Consider a *break* statement defined as follows:

$$
\begin{aligned}
&\mathit{break} : \mathit{Stmt} ::= \varepsilon\\
&\quad \mathit{code} = \mathit{goto}\ \mathit{Stmt}_1.\mathit{gotoLabel}
\end{aligned}
$$

The inherited attribute *gotoLabel* is defined by the *while* production for its *Stmt* child and other productions have semantic functions to copy this value to their *Stmt* children. By using forwarding, the *break* statement works as expected when it appears inside a *for* loop since the *code* attribute for *for* is defined by forwarding to a *while* loop construct. The *for* writer doesn't need to define, or even know about, the *gotoLabel* attribute and the *for* writer should define neither the *code* nor the *gotoLabel* attribute. This attribute represents the type of detailed semantic information the writer of the *for* construct should not need to know about.

The subtlety arises in the case when the *for* production explicitly defines the *code* attribute in terms of the *code* attribute of its children (perhaps in an attempt to generate more efficient code than that generated by the translation into a *while* loop) but doesn't define the *gotoLabel* attribute for its *Stmt* children. Since any *break* inside the *for* will need a *gotoLabel* value, this attribute should be defined by the *for* production. If we evaluate the attribute grammar with forwarding

[1] One can, however, specify attributes to be evaluated on demand in SSL.

as discussed in Section 2.2, the evaluation fails when the *Stmt* child of the *for* attempts to reference its *gotoLabel* value which should be, but isn't, defined for it by the *for* production and causes a compile time exception. If we reduce the grammar as described above, step 3 adds an incorrect definition of *gotoLabel* and this grammar passes the standard closure test but generates incorrect results. Clearly, the *for* loop must define either *both* the *code* and *gotoLabel* attributes or neither of them. Next, we describe a definedness test which can identify this type of error.

4 The Definedness Test

As we explained above, although the standard circularity test can be applied to the reduced grammar, the standard closure test is inaccurate. We propose that this problem can be solved by abandoning the closure test and modifying the circularity test, so that it encompasses both roles. It statically checks that all *required* attributes are *well defined*. That is, they have definitions and that these definitions are non-circular.

The standard circularity test [18] operates by computing a set of *dependency relations* for each nonterminal in the grammar. A dependency relation is a property of an individual abstract syntax tree which relates the root node's synthesised attributes to its inherited attributes. A synthesised attribute s is related to the inherited attribute i if the computation of s depends on the value of i. Different abstract syntax trees can have different dependency relations, even if they are of the same nonterminal type. Therefore, the circularity test computes for each nonterminal the set of all dependency relations that a tree of that type might have. Since the set of possible relations is finite, there is an algorithm which can compute them in a finite amount of time without examining every possible tree. During the process of computing these sets, it may discover that an abstract syntax tree exists in which the attributes are circularly defined.

Our definedness test replaces dependency relations with *definedness functions*. A definedness function is also a property of a particular tree and has the type $Set\ A_i \rightarrow Set\ A_s$, where $A_i(A_s)$ is the set of all inherited (synthesised) attributes. The function states which synthesised attributes can be computed on its root node if only the given set of inherited attributes is defined on the root. Consider an example definedness function w.

- For $s \in A_s$, if $s \in w(\emptyset)$, then s must have a constant value, because it does not depend on any of the inherited attributes.
- If $i \notin I, I \subseteq A_i$ and $s \in w(I), s \in A_s$, then s does not depend on i.
- If $s \notin w(A_i), s \in A_s$, then either the semantic rule for s is missing or s depends on a circular computation.

Definedness functions are very similar to the dependency relations except that they operate in opposite directions; they are given a set of inherited attributes and report which synthesised attributes can be computed. The advantage of the definedness function is that it incorporates closure as well as circularity;

for a particular tree, if the semantic function is missing for a attribute in the tree whose value is required to compute a synthesised attribute s, then s will not appear in the output of w for any given input. This is exactly the type of information we need to detect the missing *gotoLabel* semantic function when *for* explicitly defined the *code* attribute in the example above. A disadvantage of the definedness function is that it does not distinguish between circularity and closure. If s does not appear in the output of w, then this could be due to a missing semantic rule or because s depends on a circular computation. However, in both of these cases the grammar is ill-defined, so we do not see this as a major drawback.

The algorithm for the definedness test is very similar to the circularity test. The test produces a set of definedness functions for every nonterminal in the grammar. Again, since the set of possible definedness functions is finite, our algorithm uses the same technique as the circularity test to compute them in a finite amount of time without needing to examine every possible tree. A more complete description and a proof of correctness is given in a forthcoming paper by Backhouse [2].

We must note that neither the standard circularity test nor the definedness test catches a particular kind of non-termination error. It is possible to construct an infinitely large abstract syntax tree by unbounded nesting of nonterminal attributes, but no exact static test can detect this type of error.

5 Conclusion

We have introduced *forwarding* as a technique for the modular decomposition of higher-order attribute grammars. The technique is orthogonal to other features for the modular description of programming languages. Furthermore, we have demonstrated how a whole-grammar transformation can eliminate the use of forwarding altogether.

Production-valued attributes are convenient in conjunction with forwarding; their use can also be transformed away. We noted the connection with defunctionalization, and indeed it would be of interest to see whether the elimination of forwarding itself can also be understood in those terms.

If so, it would lend further credence to our belief that there is much benefit to be derived from the interaction between the functional programming and attribute grammar communities. In a separate paper, one of us (Backhouse) has shown how abstract interpretation can benefit the study of attribute grammars [2]. Conversely, Correnson, Parigot and their coworkers have argued that transformations on attribute grammars benefit functional programs [4,22].

Acknowledgements

Eric Van Wyk and Kevin Backhouse are supported by a grant from Microsoft Research. We would like to thank our colleagues both at Microsoft and at Oxford for many interesting discussions on the topic of forwarding.

References

1. S. Adams. *Modular Attribute Grammars for Programming Language Prototyping*. Ph.D. thesis, University of Southampton, 1991. 128, 135
2. K. S. Backhouse. A functional semantics of attribute grammars. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2002. 140
3. A. Carle. Hierarchical attribute grammars: Dialects, applications and evaluation algorithms. Technical Report TR93-270, Department of Computer Science, Rice University, 1993. 128
4. L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702, pages 353–369. Lecture Notes in Computer Science, 1999. 140
5. O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Third International Conference on Principles and Practice of Declarative Programming (PPDP 01)*. ACM Press, 2001. 136
6. O. de Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica*, 24(3), 2000. 135
7. O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented compilers. In *First International Symposium on Generative and Component-based Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 1999. 135
8. G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Computing Journal*, 33:164–172, 1990. 128
9. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'c: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, pages 131–144, 1996. 130
10. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 223–234. ACM Press, 1992. 128
11. H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984. 128
12. J. A. Goguen. Higher-order functions considered unnecessary for higher-order programming. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Welsey, Reading, MA, 1990. 132
13. G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*. Cambridge University Press, 1989. 128
14. Gorel Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153–172, Amsterdam, The Netherlands, 1999. INRIA rocquencourt. 128
15. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987. 135
16. S. Jones and J. Hughes. Haskell98: A non-strict purely functional language. 132
17. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994. 128

18. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–146, 1968. Corrections in 5(2):95–96, 1971. 134, 139
19. C. Le Bellec, M. Jourdan, D. Parigot, and G. Roussel. Specification and implementation of grammar coupling using attribute grammars. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag, 1993. 128
20. W Maddox. Semantically-sensitive macroprocessing. Master's thesis, The University of California at Berkeley, Computer Science Division (EECS), Berkeley, CA 94720, December 1989. 130
21. M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, 2000. 128, 130
22. D. Parigot, E. Duris, G. Roussel, and M. Jourdan. Attribute grammars: a declarative functional language. Rapport de Recherche 2662, INRIA, 1995. 140
23. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989. 137
24. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972). 136
25. Joao Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, 1999. INRIA rocquencourt. 128
26. Tim Sheard. Using metaml: A staged programming language. In *Advanced Functional Programming*, pages 207–239, 1998. 130
27. C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1. Available from URL `http://www.research.microsoft.com/research/ip/`, 1996. 128
28. S. D. Swierstra. Simple, functional attribute grammars. `http://www.cs.uu.nl/groups/ST/Software/UU_AG/`, 1999. 137
29. T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *ACM Sigplan '90 Conference on Programming Languages Design and Implementation*, pages 197–208, 1990. 128, 134
30. E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional programming: a host of language features. Technical Report PRG-RR-01-15, Computing Laboratory, University of Oxford, 2001. 128, 129, 135
31. H. Vogt. *Higher order attribute grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1989. 134
32. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *Conference on Programming Languages Design and Implementation*, pages 131–145, 1990. Published as *ACM SIGPLAN Notices*, 24(7). 128, 134
33. Daniel Weise and Roger F. Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6):156–165, 1993. 130