Remote Attribution*

John Tang Boyland**

University of California, Berkeley, Computer Science Division, Department of Electrical Engineering and Computer Science, Berkeley, CA 94720-1776, USA

Abstract. This paper introduces a method of handling direct remote attribute dependencies in attribute grammars. Once a connection between two nodes has been established, attribute values may flow in either direction. Contrary to classical attribute grammars, attribute uses and definitions may be distributed throughout the tree. Distributed definitions of a "collection" attribute are combined declaratively to yield a single definition.

Remote attribution greatly increases the expressiveness of attribute grammars, allowing attribute grammars to express computations often assumed to be possible only using side-effects or imperative constructs. However, remote attribution remains firmly grounded in the original spirit of attribute grammars because it does not require user-visible side-effects. Moreover, a safe evaluation order can be determined automatically; it is not necessary for an attribute grammar writer to manually schedule dependencies to ensure that no attribute is used before it has been evaluated. These properties are shown through a reduction to classical attribute grammars.

Attribute grammars using remote attribution can be implemented *more* efficiently than those restricted to classical attribute dependencies. Remote attribution thus provides a higher-level, more intuitive and yet still declarative notation that enables more efficient implementation.

Keywords: attribute grammars, semantic analysis

1 Introduction

Attribute grammars [16] were developed to specify the semantics of programming languages as an alternative to writing a compiler in an imperative language. An imperative program is not an ideal specification, because it expresses *how* a computation is to proceed, rather than *what* the desired result is to be. A more *declarative* approach is to specify what values need to be computed. Such a specification can be executed after first (automatically) determining an evaluation order that respects the dependencies in the specification. An advantage of declarative specifications is that they can be incrementalized by simply re-evaluating changed values. Incrementalizing an imperative program

* This research was supported in part by the Advanced Research Projects Agency (DoD) under Grant MDA972-92-J-1028, and by the National Science Foundation under Infrastructure Grant CDA-8722788. The content of the information does not necessarily reflect the position or the policy of the U.S. Government.
Last revised: Date: 1995/09/18 20:58:29

** email boyland@cs.berkeley.edu.

is much more difficult; maintaining a copy of every version of the state is impractical. Attribute grammars take the declarative approach; an attribute grammar specifies a set of *attribute equations* for each production in a context-free grammar. In the definition of classical attribute grammars, attribute values are specified as pure functions of attribute values of neighboring nodes. A variety of implementation techniques have been developed [1] and a number of usable attribute grammar implementation tools exist [4, 12, 14].

Aside from a few cases (for instance an Ada type checker [18] and a VHDL compiler [7]), however, attribute grammars have not been particularly successful either in formal language specification or in practical compiler generation. In fact, many people have the impression that attribute grammars are clumsy, often requiring longer and more complicated specifications than an imperative language would require to express the same task. In particular, classical attribute grammars require remote dependencies (that is, dependencies between nodes that do not occur in the same production instance) to be described through a chain of dependencies between neighboring nodes. Section 2 shows how this restriction clutters an attribute grammar.

This paper describes an extension to classical attribute grammars, *remote attribution*, that allows a more convenient, and indeed simpler way to define dependencies between attributes of widely separated nodes of the tree. The extension works by allowing references to nodes to be transmitted as values. At any point, attributes may be read from such nodes, thus establishing a remote dependency. Special "collection" attributes may be written through remote references. Since a node may be transported to many different locations in the tree, a collection attribute permits multiple definitions. The values of these definitions are combined into a single value; reading a collection attribute always gets this final value. Section 2 describes an attribute grammar using remote attribution that performs the same task as a classical one with little over half the complexity (22 instead of 40 attribute definitions, 12 instead of 21 attributes).

Remote attribution is declarative. As in classical attribute grammars, the attribute definitions are unordered, an evaluation order is determined solely by data dependencies. There are no user-visible side-effects; not even collection attributes "change" value. Section 3 shows how an attribute grammar with remote attribution can be converted automatically into a classical attribute grammar through the addition of "control attributes." The control attributes participate in scheduling, but can be ignored during implementation. Thus, despite the fact that remote attribution allows higher-level specifications, it does not exact a price in run-time efficiency. In fact, an implementation of an attribute grammar using remote attribution can be both more space efficient and more time efficient than an implementation of a classical attribute grammar.

2 Remote Attribution

In a classical attribute grammar, any values that need to be transmitted between remote nodes must be transmitted through the least common ancestor in the tree. Sometimes, several values may be "ready" at the same time and may be packaged up (in some sort of record) and transmitted together, but other times, packaging would cause a circular dependency and thus the values must be transmitted independently. The rules to perform the transmission are called "copy rules" because they simply copy a value through a node on the way between widely separated nodes of the tree. Classical attribute grammars are often cluttered with many copy rules. The decision of whether two values can be packaged together (thus cutting down on the number of copy rules) relies on global scheduling information. This fact forces the attribute grammar writer to schedule by hand; an implementation tool can only verify that the schedule is correct (non-circular).

With remote attribution, once a node has been transmitted elsewhere, multiple attributes may be read and written. No additional copy rules are necessary. Moreover, scheduling is done automatically, relieving the attribute grammar writer of doing the scheduling by hand. Thus remote attribution improves the ability of an attribute grammar to fulfill the task of declarative specification.

This section shows how a simple task may be accomplished in an attribute grammar, first with restricted classical attribute rules and then with the features provided by remote attribution. The task in question is *static checking* (name resolution and type-checking) for a simple programming language. *Name resolution* here refers to determining which declaration is being used for each identifier in an expression. The attribute grammars also determine when a declaration goes unused. In this example, the result of the attribution is a set of error messages.

Both attribute grammars include the complete abstract context-free grammar (shown in italics) for the programming language. Each program in the language consists of a block; each block has a sequence of variable declarations and then a sequence of statements. Variables are either of type integer or of type string. Statements are either nested blocks or assignment statements, where the expression being assigned must be of the same type as the expression being assigned to it. (Assume that the parser disallows assignments such as `1 = 2`.) Expressions can be constants or identifier uses.

2.1 Static Checking with a Classical Attribute Grammar

Figure 1 gives a classical attribute grammar for performing static checking. The first part of the attribute grammar defines the attributes for each nonterminal of the language. Each attribute is declared either *inherited* (`inh`) or *synthesized* (`syn`). Intuitively, the former attributes are "top-down" and the latter "bottom-up." Next, a set of attribute equations is defined for each production. These equations define the synthesized attributes of the result of the production and the inherited attributes of the children of the production. Furthermore, for each production, an attribute grammar may define *local attributes*. A local attribute is assigned in the same way as the other attributes.

Much of the attribute grammar in Fig. 1 concerns environments, sets of identifiers or error messages. For example, Fig. 1 contains the following rules:

decls $\rightarrow$ *decls decl*

$$\begin{aligned}
&\mathtt{decls}_1\mathtt{.envin} = \mathtt{decls}_0\mathtt{.envin}\\
&\mathtt{decl.envin} = \mathtt{decls}_1\mathtt{.envout}\\
&\mathtt{decls}_0\mathtt{.envout} = \mathtt{decl.envout}\\
&\mathtt{decls}_0\mathtt{.decld} = \mathtt{decls}_1\mathtt{.decld} \cup \mathtt{decl.decld}
\end{aligned}$$

The environment (first as $\mathtt{decls}_0\mathtt{.envin}$) must be threaded first to the list of all declarations but the last (yielding $\mathtt{decls}_1\mathtt{.envout}$) and then to the last declaration (yielding

```
program
        syn msgs
block
        inh env
        syn used, msgs
decls, decl
        inh envin
        syn envout, decld
type
        syn shape
stmts, stmt
        inh env
        syn used, msgs
expr
        inh env
        syn shape, used, msgs
-----------------------------------
program → block
        block.env = empty_env()
        program.msgs = block.msgs

block → "begin" decls stmts "end"
        local unused
        decls.envin = block.env
        stmts.env = decls.envout
        unused =
          decls.decld - stmts.used
        block.used =
          stmts.used - decls.decld
        block.msgs = stmts.msgs ∪
          if unused = {} then {}
          else {"unused: " ++ unused}
decls →
        decls.envout = decls.envin
        decls.decld = {}

decls → decls decl
        decls1.envin = decls0.envin
        decl.envin = decls1.envout
        decls0.envout = decl.envout
        decls0.decld =
          decls1.decld ∪ decl.decld

decl → id ":" type ";"
        decl.envout =
          add_env(<id,type.shape>,
                  decl.envin)
        decl.decld = {id}
```

```
type → "integer"
        type.shape = INTSHAPE
type → "string"
        type.shape = STRSHAPE

stmts →
        stmts.used = {}
        stmts.msgs = {}
stmts → stmts stmt
        stmts1.env = stmts0.env
        stmt.env = stmts0.env
        stmts0.used = stmts1.used ∪
                          stmt.used
        stmts0.msgs = stmts1.msgs ∪
                          stmt.msgs
stmt → block ";"
        block.env = stmt.env
        stmt.used = block.used
        stmt.msgs = block.msgs

stmt → expr ":=" expr ";"
        expr1.env = stmt.env
        expr2.env = stmt.env
        stmt.used = expr1.used ∪ expr2.used
        stmt.msgs =
          (if expr1.shape /= expr2.shape
           then {"type mismatch"}
           else {}) ∪ expr.msgs

expr → intconstant
        expr.shape = INTSHAPE
        expr.used = {}
        expr.msgs = {}

expr → strconstant
        expr.shape = STRSHAPE
        expr.used = {}
        expr.msgs = {}

expr → id
        local shape
        shape = lookup(id,expr.env)
        expr.shape = shape
        expr.used = {id}
        expr.msgs =
          if shape = NOT_FOUND then
            {id ++ " not declared"}
          else {}
```

Fig. 1. A classical attribute grammar for static checking

decl.envout) and finally returned as $\texttt{decls}_0\texttt{.envout}$. Additionally, the set of declared identifiers must be combined from both children. This set is used in the rules for the block to determine if any declared identifier goes unused, as well as to compute the set of "free" identifiers used in the block.

2.2 Static Checking Using Remote Attribution

In Fig. 1, the type (shape) of declaration (*decl* $\rightarrow$ *id ":" type ";"*) is only put into the environment so that the type of a variable can be determined at a use site. The type of a variable does not influence the way in which names are resolved. Conceptually, the declaration itself is what should be found and then any information needed about the declaration could be determined directly from it rather than through the environment.

Figure 2 expresses this intuition. Each declaration inserts itself into a set of declarations local to a block. Additionally a shape attribute is defined to be used remotely.

decl $\rightarrow$ *id ":" type ";"*

```
        decl.scope.decls ⊒ {<id,decl>}          remote def
        decl.shape = type.shape
⋮
```

expr $\rightarrow$ *id*

```
        local decl
        decl = lookup(id,expr.scope)
        expr.shape = decl.shape                 remote use
```

The first rule is a remote definition of a "collection attribute" decls. It adds the pair <id,decl> to the set of local declarations of the scope. At the use site, the appropriate declaration is found for the scope. The type of the expression depends on a remote attribute use of the shape attribute.

So far, remote attribution may seem to be contributing little. Not only must the type be fetched using indirection, but an extra rule is needed to place the type where it can be fetched. However, the dependencies are different. The list of declarations no longer depends on each declaration's shape. A general incremental evaluation strategy that could track remote dependencies would not have to look up every identifier again just because the type of one declaration changed [10].

Having the declaration node available at the use site makes it easier to transmit further information. In this case, determining whether a declaration is used is extremely easy to specify. Each declaration has a synthesized *collection attribute* used. A collection attribute has a starting value (here false) and a combination function (here or) used to form a final value from all the definitions:

```
decl
        ⋮
        syn used ⊒ false, (or)
```

Since the definitions are unordered, a combination function must be commutative and associative.

```
global msgs ⊒ {}, (∪)
block
        inh enclosing
        syn decls ⊒ {}, (∪)
decls, stmts, stmt
        inh scope
decl
        inh scope
        syn shape
        syn used ⊒ false, (or)
type
        syn shape
expr
        inh scope
        syn shape
```

---

program → block
```
        block.enclosing = ROOT_SCOPE
```

block → "begin" decls stmts "end"
```
        decls.scope = block
        stmts.scope = block
```

decls →

decls → decls decl
```
        decls1.scope = decls0.scope
        decl.scope = decls0.scope
```

decl → id ":" type ";"
```
        decl.scope.decls ⊒ {<id,decl>}
        decl.shape = type.shape
        msgs ⊒
          if decl.used then {}
          else {id ++ " is unused"}
```

type → "integer"
```
        type.shape = INTSHAPE
```

type → "string"
```
        type.shape = STRSHAPE
```

stmts →

stmts → stmts stmt
```
        stmts1.scope = stmts0.scope
        stmt.scope = stmts0.scope
```

stmt → block ";"
```
        block.enclosing = stmt.scope
```

stmt → expr ":=" expr ";"
```
        expr1.scope = stmt.scope
        expr2.scope = stmt.scope
        msgs ⊒
          if expr1.shape /= expr2.shape
          then {"type mismatch"}
          else {}
```

expr → intconstant
```
        expr.shape = INTSHAPE
```

expr → strconstant
```
        expr.shape = STRSHAPE
```

expr → id
```
        local decl
        decl = lookup(id,expr.scope)
        expr.shape = decl.shape
        decl.used ⊒ true
        msgs ⊒
          if decl = NOT_FOUND
          then {id ++ " not declared"}
          else {}
```

```
function lookup(id,s) =
  if s = ROOT_SCOPE then NOT_FOUND
  else let d = fetch(id,s.decls) in
    if d = NOT_FOUND then
      lookup(id,s.enclosing)
    else d
```

Fig. 2. Static checking using remote attribution

A collection attribute may be used as any other attribute. Here the declaration checks its `used` flag and generates an error message if it is unused:

decl $\rightarrow$ *id* ":" *type* ";"

```
        ⋮
        msgs ⊒ if decl.used then {} else {id ++ " is unused"}
```

Note that a use of a collection attribute will not be scheduled until all the definitions have been combined into a final value.

Collection attributes are assigned using $\sqsupseteq$ to emphasize the fact that only a partial attribute definition is being given:

expr $\rightarrow$ *id*

```
        ⋮
        decl.used ⊒ true
```

No other part of the attribute grammar handles whether declarations are used! The situation in the classical attribute grammar is far more complicated, involving many different attributes passed around almost every node in the tree. In the example using remote attribution, moreover, the fact that a declaration is unused can be determined at the declaration site, thus enabling the error to be pinpointed. Doing so in a classical attribute grammar would be very complex.

A remotely defined attribute must be a collection attribute because there can be no guarantee that it will be defined precisely once. One could permit multiple definitions as long as they were identical, and have a default for when there were no definitions, but such exceptions are messy and are insufficient to handle similar cases. Suppose instead of knowing *whether* a declaration is used, one would like to know *how many times* it is used lexically. One could attempt to increment a count for each use:

expr $\rightarrow$ *id*

```
        local decl
        decl = lookup(id,expr.env)
        decl.use_count = decl.use_count + 1  -- decl used here once
```

This intuitive definition would work in an imperative language, but as a specification, it is ill-formed. The "attribute equation" is actually an imperative (and circular!) modification of an attribute. Unfortunately, the classical attribute grammar formulation of what is expressed here is yet more complex than that for determining only *whether* a declaration is used. Using a collection attribute, the specification is both clear and declarative:

```
decl
        syn use_count ⊒ 0, (+)
⋮
```

expr $\rightarrow$ *id*

```
        ⋮
        decl.use_count ⊒ 1
```

The `use_count` attribute has a starting value of `0` and definitions are added to create the final value.

In the classical attribute grammar, error messages are collected into sets passed to the root. The collection process clutters the attribute grammar and affects productions even when they do not generate any error messages of their own. Such tasks can be accomplished using *global* collection attributes that are simply collection attributes of the root. A "sugared" notation makes these attributes easier to use. Global attributes can replace some of the imperative constructs used in attribute grammar implementations (for instance, generating error messages in Olga [11]).

Using collection attributes greatly reduces the amount of "busy-work" in attribute grammars. The attribute grammar in Fig. 2 accomplishes the same task as that in Fig. 1 at a higher level. It includes three examples of remote attribute use, `decl.shape` in the rules for $expr \to id$ and `s.decls` and `s.enclosing` in the function `lookup` and five examples of remote attribute definition, each using the $\sqsupseteq$ notation. The attribute grammar does not use user-visible side-effects.

A Default Node Reference is Needed. When an operation that returns a node, such as `lookup` in Fig. 2, cannot complete its task, it must still return something. Nontermination (sometimes signified by a "bottom" value) is unacceptable, because an undeclared identifier in the program should not cause the compiler to crash. Instead, a default node reference is needed. In the absence of conditional attribution [2], the default must permit attributes to be read and written. In Fig. 2, `NOT_FOUND` is used as a default.

Remote Definitions are Unordered. The examples in Figs. 1 and 2 are not completely comparable. The first builds up an environment as a (sequential) list, the second stores it as an unordered set with an explicit "enclosing scope pointer." Remote definition cannot be used to create a list because the definitions are unordered. However, the relative position in some arbitrary linearization of the tree such as the sequence of nodes visited in a preorder traversal may be used to check whether a declaration is defined before it is used, or whether an identifier is multiply declared.

2.3 Summary

To use remote attribution, a node is transmitted through the attribute system to another location within the tree. Attributes may be read and collection attributes may be defined for such a node. Remote attribution reduces the number of copy rules needed and also allows more scheduling decisions to be performed automatically.

3 Implementation

Efficient implementation of attribute grammars requires static analysis. However, using an attribute of a node computed dynamically, or worse, defining an attribute remotely, makes it harder for static analysis to determine a safe evaluation order. In current systems that allow something close to remote attribute use (say LIGA [14, 15]), the programmer must add "control" attributes to force the evaluation order to be safe.

This section shows that a safe order can be guaranteed *automatically* through a reduction to classical attribute grammars. This reduction is accomplished through the addition of new *fiber* attributes.[3] In the resulting classical attribute grammar, these attributes carry values from one node through multiple copy rules to a remote node. First described is the intuition behind the reduction. Next, the reduction is described more precisely.

The resulting attribute grammar may be implemented using any classical implementation method. However, at the end of this section we show that an implementation may handle remote attribution directly. In such an implementation, the fiber attributes are used only for scheduling purposes, but are ignored thereafter.

3.1 The Intuition

In an attribute grammar using remote attribution, an attribute that carries a node implicitly also carries *remote attributes*, that is any attributes that node may have. It also implicitly carries *in the reverse direction* any *partial definitions* of the collection attributes of that node. In the classical attribute grammar resulting from applying the reduction described in this section, the remote attributes and partial definitions are carried explicitly by *fiber* attributes. A fiber attribute that carries a remote attribute has the same *direction* (synthesized, inherited or local) as the attribute that carries the node to which remote attribute belongs. A fiber attribute that carries partial definitions has the opposite direction (synthesized is the opposite of inherited, local is its own opposite). For convenience, the fiber attributes that transmit remote attributes are named $A\$a$, where A is the attribute occurrence carrying the node and a is the attribute being fetched remotely. The fiber attributes that transmit partial definitions are named $A\$c\mathtt{C}$, where c is the collection attribute being remotely defined.

Intuitively, the reduction consists of a repetition of two stages; the first performs an analysis and the second adds fiber attributes. This view glosses over some messy details.

The first stage computes which attributes are fetched from or defined for nodes carried by each attribute. For example, in Fig. 4 on page 13 in the rules for $B \rightarrow$, the attribute `sn` is fetched from a node carried by `n1`. Since `n1` got its node from `B.in`, the latter attribute occurrence also has `sn` fetched from it. Working backwards through the attribute grammar, one see that `sn` is also fetched from `A.sn` and the local attribute `n`. The next remote use in `v = n2.a` leads only to determining that `a` is fetched from `n2`. Since `n2` gets its node from a remote attribute use itself, the analysis cannot yet tell where the node comes from, or which other attributes carry nodes whose `a` attribute is fetched. The remote definition of `c` is handled similarly to that of the use of `sn`.

The second stage adds fiber attributes for each needed attribute value, as determined in the first stage. For the production $B \rightarrow$, the result has more rules, but only one (new) one has a remote attribute use. This remote use appears because the analysis in the first stage was unable to determine where the node in `n2` came from.

[3] The term *fiber* is borrowed from Farrow who defines a feature with dependencies similar to remote attribute use [5].

B →

```
        local n1, n2, v, NEW: n1$sn, n2$a, n1$cC
        n1 = B.in
NEW:    n1$sn = B.in$sn
NEW:    B.in$cC = n1$cC
NEW:    n2 = n1$sn
NEW:    n2$a = n1$sn.a      remote use
NEW:    v = n2$a
NEW:    n1$cC = {v}
```

The fiber attribute `B.in$cC` has the opposite direction of the source attribute `B.in`; while the latter is inherited, the former is synthesized since it transmits a partial definition back to the node.

Since attributes fetched remotely may themselves contain nodes, it may be necessary to repeat the stages. In the example here, `n1$sn` has the attribute `a` fetched from it remotely and so it is necessary to repeat the stages. After applying the second stage again, the remote attribute use is gone and two new rules are added for this production:

```
B →
        local n1, n2, v, n1$sn, n2$a, n1$cC, NEW: n1$sn$a
        n1 = B.in
        n1$sn = B.in$sn
NEW:    n1$sn$a = B.in$sn$a
        B.in$cC = n1$cC
        n2 = n1$sn
NEW:    n2$a = n1$sn$a
        v = n2$a
        n1$cC = {v}
```

At this point, in our example, there are no more remote attribute uses or definitions and the transformation is complete. Unfortunately, in general, the process may not converge. If an attribute of node contains a node of the same "type," it is possible that the process may continually add new fiber attributes. In the larger example given in Fig. 2, nontermination results from the fact that the enclosing scope attribute of a block refers to another block and that the `lookup` function can search an unbounded number of nested scopes. Therefore, it is necessary to cut off the repetition at some point.

3.2 Fiber Analysis

The preceding section gave the intuition involved in removing remote attribution. This section describes the reduction more precisely, as an analysis followed by two transformations. For the sake of brevity, the analysis of functions calls is omitted. Although the analysis in its pure form still suffers from nontermination, "approximations" may be defined that ensure termination.

Doing all the analysis at once requires that instead of merely determining which *attributes* are fetched from nodes carried by attributes, it is necessary to determine which *access paths* are applied to such nodes. Each access path of attributes applied to a node is called a *fiber*. In the access path comprising a fiber, "." is replaced by "_" in order to avoid confusion when generating fiber attribute names.

Figure 3 describes fiber analysis and the first part of the reduction to a classical attribute grammar. In order to simplify the presentation, it is assumed that if a node is used as a value, then any collection attributes it may have are among its *defined attribute occurrences* (a synthesized attribute of the nonterminal of the production or an inherited attribute of a child). It is also assumed that local attributes are introduced as necessary so that each rule has one of the forms used in the figure, where A is a local attribute, B is a local attribute or a *used attribute occurrence* (an inherited attribute of the nonterminal of the production or a synthesized attribute of a child) a is any attribute and c is a collection attribute.

Some of the fiber attributes added are collection attributes, but after the first part of the reduction, no collection is defined remotely. As a result, a classical attribute definition can be formed by combining all statically visible assignments. The final effect of both of these transformations can be seen in Fig. 4.

The specification may lead to infinite sets, but termination can be ensured through the use of a fiber approximation: a finite partition of the infinite set of possible fibers. Modifying the analysis to handle approximations is straightforward. The resulting analysis is coarser and the reduction may yield a circular attribute grammar. Farrow investigated approximation for remote attribute use only and proved that no finite approximation will work for all noncircular attribute grammars [5]. A simple approximation that collapses attribute access cycles yields a non-circular classical attribute grammar when applied to the attribute grammar in Fig. 2 (in fact the result is OAG). Our experience indicates that such approximations can normally be found.

3.3 "Remote-Aware" Implementation

In an implementation handling remote attribution directly, the fiber attribute are treated as *control attributes*: used for scheduling purposes, but ignored thereafter.[4] In the resulting implementation, actual pointers to nodes may be transmitted through the attribute evaluation mechanism. Fields representing attributes may be read or written indirectly using the pointers. Stated another way, code is only generated for definitions in the source attribute grammar!

Such an implementation is much more efficient than a classical implementation of the transformed attribute grammar. Moreover, since remote attribute definition can be implemented using simple field modifications, collection attributes can be implemented as efficiently as the equivalent stores of an imperative language. A hand-written classical attribute grammar performing a task most naturally expressed using collection attributes will have a much less efficient implementation.

[4] The remote-aware implementation of remote attribution must be able to handle non-normal-form attribute definitions. Kasten's OAG algorithm [13] would work correctly on the attribute grammar after normalization because it schedules each attribute as soon as possible, but not all schedulers have this property.

Notation:
$A(X)$ $(C(X))$: (collection) attributes of X.
$U(X_j)$ $(D(X_j))$: used (defined) attr. occs.
$S \sqsupseteq^F T \iff S \supseteq \{f \mid f \in T, \mathrm{forward}(f), f = f'\_x, f' \in S\}$
$S \sqsupseteq^B T \iff S \supseteq \{f \mid f \in T, \neg\mathrm{forward}(f), f = f'\_x, f' \in S\}$
$f \in^S F \iff (f \in F \wedge f \neq \epsilon \wedge \mathrm{same}(f))$
$f \in^C F \iff (f \in F \wedge f \neq \epsilon \wedge \neg\mathrm{same}(f))$
$\mathrm{same}(f) \iff \mathrm{even}(\mathrm{count}(c\mathtt{C}, f))$
$\mathrm{forward}(f\_x) \iff \neg\mathrm{same}(f)$
`+` means generate this fiber attribute equation
`!` means replace the equation with this one

$X\ dir\ a$
  $F(a) \ni \epsilon$
  + $X\ dir\ a\$f \quad (f \in^S F(a))$
  + $X\ dir^{-1}\ a\$f \quad (f \in^C F(a))$

$X_0 \to X_1 \ \ldots \ X_n$
  $F(X_j.a) \ni \epsilon, \quad a \in A(X_j)$
  $F(a) \sqsupseteq^B F(X_j.a), \quad a \in U(X_j)$
  $F(X_j.a) \sqsupseteq^F F(a), \quad a \in U(X_j)$
  + $\mathtt{local}\ X_j\_a\$f \quad (a \in D(X_j))$
    $(f \in^S F(X_j.a) \vee f \in^C F(X_j.a))$

$\mathtt{local}\ A$
  $F(A) \ni \epsilon$
  + $\mathtt{local}\ A\$f$
    $(f \in^S F(A) \vee f \in^C F(A))$

$A = B$
  $F(B) \sqsupseteq^B F(A)$
  $F(A) \sqsupseteq^F F(B)$
  + $A\$f = B\$f \quad (f \in^S F(A) \cap F(B))$
  + $B\$f \sqsupseteq A\$f \quad (f \in^C F(B) \cap F(A))$

$X_j.a = B \quad (a \in D(X_j), a \notin C(X_j))$
  $F(B) \sqsupseteq^B F(a) \cup F(X_j.a)$
  $F(a) \sqsupseteq^F F(B)$
  $F(X_j.a) \sqsupseteq^F F(B)$
  $\mathtt{for}\ f \in F(B)$
    + $X_j.a\$f = B\$f \ (f \in^S F(a))$
    + $X_j\_a\$f = B\$f \ (f \in^S F(X_j.a)$
    + $B\$f \sqsupseteq X_j.a\$f \ (f \in^C F(a))$
    + $B\$f \sqsupseteq X_j\_a\$f \ (f \in^C F(X_j.a))$

$A = X_j$
  $\mathtt{for}\ a \in A(X_j)$
    $F(X_j.a) \sqsupseteq^B \{f \mid a\_f \in F(A)\}$
    $F(A) \sqsupseteq^F \{a\_f \mid f \in F(X_j.a)\}$
    + $A\$a = X_j.a \ (a \in^S F(A))$
    $\mathtt{if}\ a \in U(X_j)$
      + $A\$a\_f = X_j.a\$f$
        $(a\_f \in^S F(A), f \in^S F(a))$
      + $X_j.a\$f \sqsupseteq A\$a\_f$
        $(f \in^C F(a), a\_f \in^C F(A))$
    $\mathtt{else}$
      + $A\$a\_f = X_j\_a\$f$
        $(a\_f \in^S F(A), f \in^S F(X_j.a))$
      + $X_j\_a\$f \sqsupseteq A\$a\_f$
        $(f \in^C F(X_j.a), a\_f \in^C F(A))$
  $\mathtt{for}\ c \in C(X_j)$
    $\mathtt{assert}\ c \in D(X_j)$
    $F(A) \sqsupseteq^F \{c\mathtt{C}\_f \mid f \in F(c) \cup F(X_j.c)\}$
    $F(c) \sqsupseteq^F \{f \mid c\mathtt{C}\_f \in F(A)\}$
    $F(X_j.c) \sqsupseteq^F \{f \mid c\mathtt{C}\_f \in F(A)\}$
    + $X_j.c \sqsupseteq \mathtt{collect}(A\$c\mathtt{C}, start_c, op_c)$
      $(c\mathtt{C} \in^S F(A))$
    $\mathtt{for}\ c\mathtt{C}\_f \in F(A))$
      + $X_j.c\$f \sqsupseteq A\$c\mathtt{C}\_f \ (f \in^S F(c))$
      + $X_j\_c\$f \sqsupseteq A\$c\mathtt{C}\_f \ (f \in^S F(X_j.c))$
      + $A\$c\mathtt{C}\_f \sqsupseteq X_j.c\$f \ (f \in^C F(c))$
      + $A\$c\mathtt{C}\_f \sqsupseteq X_j\_c\$f \ (f \in^C F(X_j.c))$

$A = B.a$
  $F(B) \sqsupseteq^B \{a\_f \mid f \in F(A)\}$
  $F(A) \sqsupseteq^F \{f \mid a\_f \in F(B)\}$
  ! $A = B\$a$
  + $A\$f = B\$a\_f$
    $(f \in^S F(A), a\_f \in^S F(B))$
  + $B\$a\_f \sqsupseteq A\$f$
    $(a\_f \in^C F(B), f \in^C F(A))$

$A.c \sqsupseteq B$
  $F(B) \sqsupseteq^B \{f \mid c\mathtt{C}\_f \in F(A)\}$
  $F(A) \sqsupseteq^B \{c\mathtt{C}\_f \mid f \in F(B)\}$
  ! $A\$c\mathtt{C} \sqsupseteq \{B\}$
  + $A\$c\mathtt{C}\_f \sqsupseteq B\$f$
    $(c\mathtt{C}\_f \in^C F(A), f \in^S F(B))$
  + $B\$f \sqsupseteq A\$c\mathtt{C}\_f$
    $(f \in^C F(B), c\mathtt{C}\_f \in^S F(A))$

Fig. 3. Fiber Analysis

```
S
        syn out
A
        syn sn, a, s
        syn c ⊒ 0, (+)
B
        inh in
```

```
S → A B
        B.in = A.sn
        S.out = A.s

A →
        local n
        n = A
        A.sn = n
        A.a = 0
        A.s = A.c

B →
        local n1, n2, v
        n1 = B.in
        n2 = n1.sn        remote use
        v = n2.a          remote use
        n1.c ⊒ v          remote def
```

$F(\texttt{out}) = F(\texttt{a}) = F(\texttt{s}) = F(\texttt{c}) = \{\epsilon\}$
$F(\texttt{sn}) = F(\texttt{in}) = \{\epsilon\text{,sn,sn\_a,cC}\}$
$S \to A\ B$
$\quad F(\texttt{S.out}) = F(\texttt{B.in}) = \{\epsilon\}$
$\quad F(\texttt{A.a}) = F(\texttt{A.s}) = \{\epsilon\}$
$\quad F(\texttt{A.c}) = \{\epsilon\}$
$\quad F(\texttt{A.sn}) = \{\epsilon\text{,sn,sn\_a,cC}\}$
$A \to$
$\quad F(\texttt{A.a}) = F(\texttt{A.s}) = \{\epsilon\}$
$\quad F(\texttt{A.c}) = \{\epsilon\}$
$\quad F(\texttt{A}_3\texttt{.sn}) = \{\epsilon\text{,a}\}$
$\quad F(\texttt{n}) = \{\epsilon\text{,sn,sn\_a,cC,a}\}$
$B \to$
$\quad F(\texttt{B.in}) = F(\texttt{n1}) = \{\epsilon\text{,sn,sn\_a,cC}\}$
$\quad F(\texttt{n2}) = \{\epsilon\text{,a}\}$
$\quad F(\texttt{v}) = \{\epsilon\}$

```
S
        syn out
A
        syn sn, sn$sn, sn$sn_a
        inh sn$cC
        syn a, s, c
B
        inh in, in$sn, in$sn_a
        syn in$cC
```

```
S → A B
        B.in = A.sn
        B.in$sn = A.sn$sn
        B.in$sn_a = A.sn$sn_a
        A.sn$cC = B.in$cC
        S.out := A.s
A →
        local A_sn$a, n, n$sn,
              n$sn_a, n$cC, n$a
        n = A
        n$sn = A.sn
        n$sn_a = A_sn$a
        n$a = A.a
        A.c = collect(n$cC,0,(+))
        A.sn = n
        A.sn$sn = n$sn
        A.sn$sn_a = n$sn_a
        A_sn$a = n$a
        n$cC = A.sn$cC
        A.a = 0
        A.s = A.c
B →
        local n1, n1$sn, n1$sn_a, n1$cC,
              n2, n2$a, v
        n1 = B.in
        n1$sn = B.in$sn
        n1$sn_a = B.in$sn_a
        B.in$cC = n1$cC
        n2 = n1$sn
        n2$a = n1$sn_a
        v = n2$a
        n1$cC = {v}
```

Fig. 4. A simple attribute grammar using remote attribution, the results of fiber analysis and the reduction to a classical attribute grammar

4 Related Work

LIGA [14], part of the Eli compiler construction system [8], has a notation that can reduce the number of copy rules. The `INCLUDING` construct allows a rule to refer to the closest ancestor node that has a certain attribute defined; the corresponding `CONSTITUENTS` construct allows a rule to refer to attributes of descendants. These features are somewhat orthogonal to remote attribution because they avoid copy rules between statically determined nodes, whereas remote attribution allows direct dependencies between nodes found during attribution.

LIGA also supports control attributes allowing an attribute grammar to make use of, say, an imperative interface to a symbol table. Control attributes represent preconditions and postconditions for scheduling imperative actions, such as inserting an element in a symbol table [15]. However, these control attributes must be written into the attribute grammar by the compiler writer, and omitting a control attribute *may* lead to an incorrect schedule. For example, an identifier may be looked up in a symbol table before all the declarations have been inserted.

Johnson and Fischer define a construct called a non-local attribute grammar including a feature analogous to remote attribute use [10]. They also describe an algorithm for producing a classical (local) attribute grammar from a non-local one using user-written annotations. It appears that these annotations are not powerful enough to describe the results of fiber analysis using approximations such as the one defined by Farrow. Neither do non-local attribute grammars give the power of remote attribute definition.

Görel Hedin defines an extension called door attribute grammars [9]. Like Johnson and Fischer, she is interested in making incremental re-evaluation of attribute grammars more efficient. In the extension, remote attribute use is restricted to special tree annotations called "door objects." There is no automatic analysis of remote dependencies, rather "visit procedures" for door objects must be written by hand. In door attribute grammars, a collection may be transmitted to multiple locations in the tree where objects may be inserted into it. This feature comes close to remote attribute definition in power. Again, manual decisions are needed to ensure correct implementation.

Another extension of attribute grammars that involves passing nodes through the attribute system is higher-order attribute grammars [17, 19]. In a higher-order attribute grammar, rules may construct new subtrees and graft them into the tree being attributed. At no point, however, may attributed nodes be transmitted through the attribute system.

Composable attribute grammars [6] and their extension, composable tree attributions [3] have a feature roughly equivalent to remote attribute use. In these systems it is possible to construct new nodes and pass them through the attribute system, possibly fetching attribute values. If this feature is used to do more than simply package attribute values, however, the attribute grammar, in the terminology of composable attribute grammars, is "non-separable." The published references of these two systems require descriptional composition for implementation. The analysis given in Sect. 3 was based on analysis given by Farrow for handling just such situations [5]. Something similar to collection attributes may be defined, but direct remote attribute definition is not permitted in these systems.

Acknowledgements

This paper is an outgrowth of a discussion between William Maddox and myself. Rodney Farrow kindly sent a report on analysis of a similar problem; the report was very valuable for clarifying fiber approximation. Nicholas Weaver, Tim Wagner, Manuel Fähndrich, Susan Graham, Rodney Farrow and William Maddox provided useful criticism on several drafts of this paper.

References

1. ALBLAS, H. Attribute evaluation methods. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings, Prague, Czechoslovakia, 4-13 June 1991* (1991), H. Alblas and B. Melichar, Eds., vol. 545 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 48–113.
2. BOYLAND, J. Conditional attribute grammars. Submitted to ACM TOPLAS, 1995.
3. BOYLAND, J., AND GRAHAM, S. L. Composing tree attributions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1994), pp. 375–388.
4. FARROW, R. The Linguist translator-writing system. Tech. rep., Declarative Systems Inc., June 1989.
5. FARROW, R. Fibered evaluation in Linguist. unpublished, 1990.
6. FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the Nineteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1992), pp. 223–234.
7. FARROW, R., AND STANCULESCU, A. G. A VHDL compiler based on attribute grammar methodology. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (June 1989), pp. 120–130.
8. GRAY, R. W., HEURING, V. P., LEVI, S. P., SLOANE, A. M., AND WAITE, W. M. Eli: A complete flexible compiler construction system. *Communications of the ACM 35*, 2 (Feb. 1992), 121–131.
9. HEDIN, G. An overview of door attribute grammars. In *Proceedings of Compiler Construction, 5th International Conference, CC'94* (Apr. 1994), P. A. Fritzson, Ed., vol. 786 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 31–51.
10. JOHNSON, G. F., AND FISCHER, C. N. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages* (Jan. 1985), pp. 141–151.
11. JOURDAN, M., BELLEC, C. L., AND PARIGOT, D. The OLGA attribute grammar description language: Design, implementation and evaluation. In *Attribute Grammars and their Applications. International Conference WAGA Proceedings, Paris, France, 19-21 Sept. 1990* (1990), P. Deransart and M. Jourdan, Eds., vol. 461 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 222–237.
12. JOURDAN, M., AND PARIGOT, D. Internals and externals of the FNC-2 attribute grammar system. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings, Prague, Czechoslovakia, 4-13 June 1991* (1991), H. Alblas and B. Melichar, Eds., vol. 545 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 485–504.
13. KASTENS, U. Ordered attributed grammars. *Acta Informatica 13*, 3 (Mar. 1980), 229–256.

14. KASTENS, U. Attribute grammars in a compiler construction environment. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings, Prague, Czechoslovakia, 4-13 June 1991* (1991), H. Alblas and B. Melichar, Eds., vol. 545 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 380–400.
15. KASTENS, U., AND WAITE, W. M. Modularity and reusability in attribute grammars. *Acta Informatica 31* (1994), 601–627.
16. KNUTH, D. E. Semantics of context free languages. *Math Systems Theory 2*, 2 (June 1968), 127–145. Errata *Math Systems Theory 5*, 1 (1971), 95–96.
17. SWIESTRA, D., AND VOGT, H. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings, Prague, Czechoslovakia, 4-13 June 1991*, H. Alblas and B. Melichar, Eds., vol. 545 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991, pp. 256–296.
18. UHL, J., DROSSOPOULOU, S., PERSCH, G., GOOS, G., DAUSMANN, M., WINTERSTEIN, G., AND KIRCHGÄSSNER, W. *An Attribute Grammar for the Semantic Analysis of Ada*, vol. 139 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1982.
19. VOGT, H. H., SWIESTRA, S. D., AND KUIPER, M. F. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (June 1989), pp. 131–145.