

# Remote Attribute Grammars

JOHN TANG BOYLAND

*University of Wisconsin-Milwaukee*

**Abstract.** Describing the static semantics of programming languages with attribute grammars is eased when the formalism allows direct dependencies to be induced between rules for nodes arbitrarily far away in the tree. Such *direct non-local* dependencies cannot be analyzed using classical methods, which enable efficient evaluation.

This article defines an attribute grammar extension (“remote attribute grammars”) to permit references to objects with fields to be passed through the attribute system. Fields may be read and written through these references. The extension has a declarative semantics in the spirit of classical attribute grammars. It is shown that determining circularity of remote attribute grammars is undecidable.

The article then describes a family of conservative tests of noncircularity and shows how they can be used to “schedule” a remote attribute grammar using standard techniques. The article discusses practical batch and incremental evaluation of remote attribute grammars.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages

Additional Key Words and Phrases: Language description, remote attribution, collection attributes

## 1. Introduction

Attribute grammars [Knuth 1968] 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 reevaluating changed values. Incrementalizing an imperative program 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 clas-

---

This work has been supported in part by the National Science Foundation (CCR-9984681) and the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

Author’s address: Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, P.O. Box 784, Milwaukee, WI 53201, email: john.boyland@acm.org.

©ACM, 2005. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Journal of the ACM, 52, 4, (July 2005) <http://doi.acm.org/10.1145/TBD>

sical attribute grammars, attribute values are specified as pure functions of attribute values of neighboring nodes. A variety of implementation techniques have been developed [Alblas 1991] and several usable attribute grammar implementation tools exist [Farrow 1989; Jourdan and Parigot 1991; Kastens 1991].

Attribute grammars have been used successfully in a few cases (for instance an Ada type checker [Uhl et al. 1982] and a VHDL compiler [Farrow and Stanculescu 1989]), but many people seem to have the impression that attribute grammars are clumsy—more complicated than the equivalent imperative programs. In particular, classical attribute grammars require dependencies between nodes that do not occur in the same production instance to be described through a chain of dependencies between neighboring nodes. The LIGA attribute grammar system [Kastens 1991] includes features that address this limitation, enabling, for example, a rule for a node to refer directly to an ancestral node by type. However, even this system does not schedule attribute grammars with dependencies between unrelated nodes in the tree.

Starting with Johnson and Fischer’s proposal for “non-local” attribute grammars [Johnson 1983; Johnson and Fischer 1985], several researchers have investigated the possibility of permitting direct non-local dependencies in attribute grammars. In Johnson’s work, the non-local dependencies are established through opaque semantic rules. In some more recent work, the dependencies are induced by passing objects with fields through the attribute system. Elsewhere, the fields may be read, and (depending on the extension) other fields may be written. Since the places in the tree where the fields are written may be far removed from the places where they are read, these abilities are called *remote attribution* [Boyland 1996b]. Direct non-local dependencies increase the expressive power of attribute grammars [Maddox 1997], reduce the number of “copy rules” and permit more efficient incremental implementation [Hedin 1994; Boyland 2002].

A related technique is that of “higher-order attribution” in which attribute rules may create trees that may be rooted at various points and attributed there. Section 7 compares the closely related concepts of remote attribution and higher-order attribute grammars; the most important distinction being that higher-order attribute grammars are purely functional so that the generated trees do not have identity.

*Remote* (direct non-local) dependencies require extensions to the standard attribute grammar analyses. If objects are added with only trivial extension to the analysis, then, depending on how the extension is formulated, standard analysis may either report spurious circularities, or (worse) fail to notice dependencies at all. In the latter case, we say that the analysis is *unsafe*. For example, in Augusteijn’s Elegant system [Augusteijn 1990], a field in an object can be designated “lazy,” thus permitting the object to be passed through the attribute system before the field has been assigned. No analysis ensures that the field is assigned before it is used. With LIGA [Kastens 1991], objects can be created in multiple passes, but the description writer must direct the scheduling through the addition of dependencies involving extra *control attributes*. The implementation tool is unable to determine if the extra dependencies are sufficient. Hedin’s door attribute grammars [Hedin 1994] leave responsibility for the scheduling of remote dependencies to hand-written code. Her “reference attributed grammars” [Hedin 2000] use demand-driven evaluation to ensure a safe schedule. Johnson gives an optimal incremental reevaluation algorithm for non-local attribute grammars which satisfy a “node-wise non-circularity” property for which no algorithm is given. The property states that no information read through a non-local de-

dependency travels back to its source node through traditional local dependencies, and vice versa. This restriction rules out some useful idioms such as marking which declarations are used through a non-local dependency.

Static scheduling of remote attribute grammars has been described in earlier work [Boylard 1998]. “Control attributes” are generated that ensure that fields are defined before they are used. *Control attributes* carry no value, have no run-time significance, and are used merely to constrain the schedule. This article extends this earlier work with an improved analysis, greater rigor, proofs of correctness and descriptions of additional experience.

The first section following this introduction gives a flavor of the expressive power provided by remote attribution. The next section describes our extended attribute grammars precisely. Section 4 describes a reduction to (improper) classical attribute grammars that is then used in Section 5 to build algorithms that enable static scheduling of remote attribute grammars. Section 6 describes some practical experience with versions of the algorithms. Section 7 reviews related extensions and analyses. The article concludes with some suggestions for further research.

## 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 (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. Independent transmission increases the number of rules for an attribute grammar and, in the case of name resolution, may require multiple lookups with the same key in isomorphic dictionaries. The decision of whether two values can be packaged together (thus reducing complexity and increasing efficiency) relies on global scheduling information, and thus should be left to an implementation tool, not the description writer.

In this section, we describe an extension in which objects with fields may be created. References to these objects may be transmitted through the attribute system. When a reference to an object reaches any location in the tree, its fields may be read and written. Copy rules may be necessary to transmit references, but no additional copy rules are necessary. The fields of an object are scheduled automatically and safely, thus relieving the attribute grammar writer of doing the scheduling by hand. In this way 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 whether 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 slanted font) 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 are constants or identifier uses.

2.1. **STATIC CHECKING WITH A CLASSICAL ATTRIBUTE GRAMMAR.** Figure 1 gives a classical attribute grammar for performing static checking from earlier work [Boyland 1998]. 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.” A set of (oriented) attribute equations is specified 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 *locals* (also known as “local attributes”). A local is assigned in the same way as an attribute.

Much of the attribute grammar in Figure 1 concerns environments, sets of identifiers or error messages. These rules make the attribute grammar more complex, although practical attribute grammar systems such as LIGA [Kastens 1991] and FNC2 [Jourdan and Parigot 1991] have ways of expressing these rules more succinctly. More seriously, passing around sets of used identifiers and looking up identifiers in them implies an inefficiency that would not arise in an imperative description that passed around objects and set a *used* flag in the object when it is used.

2.2. **STATIC CHECKING USING REMOTE ATTRIBUTION.** In Figure 1, the type (*shape*) of a 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, (a reference to) the object representing a declaration itself is what should be found and then any information needed about the declaration could be determined directly from the object rather than through the environment.

Figure 2 expresses this intuition. Each declaration inserts an object into the set of declarations local to its block. This object has two fields: *shape* records the type of the declaration, and *used* records whether the variable declared is used anywhere. At the use site, the appropriate declaration is found in the declarations for the scope. The type of the expression depends on a use of the *shape* attribute of an object created remotely.

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 create the object to 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.

Having the object representing the declaration 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. The object created for each declaration has a *collection field* named *used*. A collection field has a starting value (here *false*) and a combination function (here *or*) used to form a final value from all the definitions. Since the definitions are unordered, a combination function must be commutative and associative.

Collection fields are defined using  $\sqsubseteq$  to emphasize the fact that only a partial attribute definition is being given:

$expr \rightarrow id$

```

program                               type → "integer"
  syn msgs                             type.shape = INTSHAPE
block, stmts stmt                     type → "string"
  inh env                               type.shape = STRSHAPE
  syn used, msgs
decls, decl                           stmts →
  inh envin, uin                       stmts.used = {}
  syn envout, uout, msgs               stmts.msgs = {}
type                                   stmts → stmts stmt
  syn shape                             stmts1.env = stmts0.env
  inh env                               stmt.env = stmts0.env
  syn shape, used, msgs                 stmts0.used = stmts1.used ∪
                                          stmt.used
program → block                       stmts0.msgs = stmts1.msgs ∪
  block.env = empty_env()              stmt.used
  program.msgs = block.msgs            stmt.msgs
block → "begin" decls stmts "end"     stmt → block ";"
  decls.envin = block.env              block.env = stmt.env
  stmts.env = decls.envout             stmt.used = block.used
  decls.uin = stmts.used               stmt.msgs = block.msgs
  block.used = decls.uout             stmt → expr " := " expr ";"
  block.msgs =                         expr1.env = stmt.env
    decls.msgs ∪ stmts.msgs           expr2.env = stmt.env
  decls →                               stmt.used = expr1.used ∪ expr2.used
  decls.envout = decls.envin           stmt.msgs =
  decls.uout = decls.uin               (if expr1.shape /= expr2.shape
  decls.msgs = {}                      then {"type mismatch"}
  decls → decls decl                   else {}) ∪ expr1.msgs ∪ expr2.msgs
  decls1.envin = decls0.envin       expr → intconstant
  decl.envin = decls1.envout         expr.shape = INTSHAPE
  decls0.envout = decl.envout         expr.used = {}
  decl.uin = decls0.uin              expr.msgs = {}
  decls1.uin = decl.uout             expr → strconstant
  decls0.uout = decls1.uout         expr.shape = STRSHAPE
  decls0.msgs =                       expr.used = {}
  decls1.msgs ∪ decl.msgs           expr.msgs = {}
decl → id ":" type ";"                expr → id
  decl.envout =                         local shape
  add_env(<id,type.shape>,              shape = lookup(id,expr.env)
  decl.envin)                          expr.shape = shape
  decl.uout = decl.uin \ id            expr.used = {id}
  decl.msgs =                           expr.msgs =
  (if id ∈ decl.uin then {}           if shape = NOT_FOUND then
  else {"unused: " ++ id})            {id ++ " not declared"}
                                          else {}

```

FIG. 1. A classical attribute grammar for static checking

```

global msgs  $\sqsubseteq$  {} with (U)           type  $\rightarrow$  "integer"
                                         type.shape = INTSHAPE
block, decls, decl, stmts, stmt      type  $\rightarrow$  "string"
inh scope                             type.shape = STRSHAPE
type
  syn shape                             stmts  $\rightarrow$ 
expr
  inh scope                             stmts  $\rightarrow$  stmts stmt
  syn shape                             stmts1.scope = stmts0.scope
                                         stmt.scope = stmts0.scope

program  $\rightarrow$  block
  block.scope = ROOT_SCOPE           stmt  $\rightarrow$  block ";"
                                         block.scope = stmt.scope

block  $\rightarrow$  "begin" decls stmts "end"
  local contour =
    { ds  $\sqsubseteq$  {} with (U),
      encl = block.scope }
  decls.scope = contour
  stmts.scope = contour
  stmt  $\rightarrow$  expr ":@" expr ";"
                                         expr1.scope = stmt.scope
                                         expr2.scope = stmt.scope
                                         if expr1.shape  $\neq$  expr2.shape
                                         then msgs  $\sqsubseteq$  {"type mismatch"}
                                         endif

decls  $\rightarrow$ 
  expr  $\rightarrow$  intconstant
                                         expr.shape = INTSHAPE
decls  $\rightarrow$  decls decl
  decls1.scope = decls0.scope      expr  $\rightarrow$  strconstant
  decl.scope = decls0.scope         expr.shape = STRSHAPE

decl  $\rightarrow$  id ":" type ";"           expr  $\rightarrow$  id
  local d =
    { shape = type.shape,
      used  $\sqsubseteq$  false with (or)}
  decl.scope.ds  $\sqsubseteq$  {<id,d>}
  if not d.used then
    msgs  $\sqsubseteq$  {id ++ " is unused"}
  endif
  local d
  d = lookup(id,expr.scope)
  expr.shape = d.shape
  if d = NOT_FOUND then
    msgs  $\sqsubseteq$  {id ++ " not declared"}
  else
    d.used  $\sqsubseteq$  true
  endif

function lookup(id,s)
  if s = ROOT_SCOPE then
    result = NOT_FOUND
  else
    local d = fetch(id,s.ds)
    if d = NOT_FOUND then
      result = lookup(id,s.encl)
    else
      result = d
    endif
  endif
endif

```

FIG. 2. Static checking using remote attribution

```

:
  d.used  $\sqsupseteq$  true

```

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

```

decl  $\rightarrow$  id ":" type ";"
:
if not d.used then
  msgs  $\sqsupseteq$  {id ++ " is unused"}
endif

```

(This example uses conditional attribution [Boyland 1996a].) A use of a collection field will not be scheduled until all the definitions have been combined into a final value. 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.

This simple remote attribute grammar also serves as an example of a “node-wise circular” attribute grammar that could not be handled by Johnson’s [Johnson 1983] non-local attribute scheduler. The problem is that the existence of the declaration is known by the scope coming down to the use from its parents, but then the fact that it is used is sent through the remote reference. From the point of view of the declaration, a compound dependency path travels through its own tree parent and returns through the remote dependency. The effect of this path cannot be captured by either an OI summary dependency graph for its parent or by an analogous summary through the remote object. This limitation apparently did not cause problems because Johnson’s description of non-local attribute grammars never mentions any equivalent of collection fields.

A remotely defined field must be a collection field 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, for example as an aid to register allocation. One could attempt to increment a count for each use:

```

expr  $\rightarrow$  id
  local d
  d = lookup(id, expr.env)
  d.use_count = d.use_count + 1 -- Error!

```

This intuitive definition would work in an imperative language, but as a specification, it is ill-formed. The “equation” is actually an imperative modification of an field. 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 field, the specification is both clear and declarative:

```

  d = { shape = type.shape,
        use_count  $\sqsupseteq$  0 with (+) }
:
  d.use_count  $\sqsupseteq$  1

```

The `use_count` field 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 collections* that are simply collection fields of a special global object created in the rules for the root production. A “sugared” notation makes these collection fields easier to use. Global collections generalize the error message feature of Olga [Jordan et al. 1990].

Using remote attribution greatly reduces the amount of “busy-work” in attribute grammars. Previous extensions for reducing copy rules (such as the `INCLUDING` keyword in LIDO [Kastens 1991]) can further simplify both examples; for example, to directly copy the environment/scope from a block node to identifier occurrences in expressions. But by leaving the scheduling of object construction to the implementation tool, the attribute grammar in Figure 2 accomplishes the same task as that in Figure 1 at a higher level, and it does so without user-visible side-effects.

The examples in Figures 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. An advantage of the unordered nature of remote collections is that they are more naturally incrementalized: a new element can be added, or an element can be removed without disturbing what remains the same. This advantage is exploited in efficient incremental evaluators for remote attribute grammars [Boyland 2002].

On the other hand, in languages with strict definition-before-use rules (such as Pascal), using unordered collections may cause certain attribute rules (such as for the evaluation of named constants) to become circular. In such cases, the attribute grammar writer using a formalism that permits remote references must decide between the convenience of unordered collections and the need for non-circular evaluation.

2.3. CREATING AND TRAVERSING CYCLIC STRUCTURE. In remote attribution the *creation* of objects is decoupled from the assigning of fields of the object. As a result, it is straightforward to construct cyclic structures. For example, when performing static checking of programs in an object-oriented language such as Java, an entry object is created for each class declaration. This object will have a field indicating the superclass; it will be a reference to the entry object for the superclass. Of course, in Java (as in most object-oriented languages) a class cannot be its own superclass (even indirectly), but this possibility may arise in erroneous but syntactically correct Java programs. Thus a cyclic structure may be created.

Furthermore, a class may have fields that refer to instances of this class (a recursive type). This situation may lead to more complex cyclic structures, depending on how one wishes to represent classes and fields for static checking purposes. Remote Attribute Grammars enables the creation of such cyclic structures, which makes the process of describing the “static semantics” of an language more succinct and straightforward.

The (potentially) circular links may be traversed by object methods (as in Hedin’s Ref-

$S \rightarrow A B$ $B.i = A.s$ $A.j = B.t$ $S.x = A.r$ $A \rightarrow a$ $A.s = g()$	$A.r = h(A.j)$ $B \rightarrow b$ $B.t = f1(B.i)$ $B \rightarrow c$ $B.t = f2()$
--	---

FIG. 3. A simple attribute grammar

erence Attribute Grammars [Hedin 2000] or Maddox’s Colander system [Maddox 1997]) or used directly in a *circular* remote attribute grammar [Boyland 1996b; Magnusson and Hedin 2003].

2.4. SUMMARY. To use remote attribution, a reference to an object is transmitted through the attribute system to another location within the tree. Fields may be read from, and collection fields may be defined for the object through the reference. Remote attribution reduces the number of attribute rules needed and also allows more scheduling decisions to be performed automatically. The following section formalizes these concepts.

### 3. DEFINITIONS

This section first reviews classical attribute grammars. It then defines remote attribute grammars as an extension. This article uses similar classical definitions as in earlier work [Boyland 1996a], the main difference being the addition of local attributes.

3.1. CLASSICAL ATTRIBUTE GRAMMARS. A *context-free grammar* is a tuple with four finite sets  $(N, T, Z, P)$ :  $N$  is the set of *nonterminals*;  $T$  is the set of *terminals* ( $N \cap T = \emptyset$ );  $Z \in N$  is the *start* symbol; and  $P$  is the set of *productions*. Define  $\Sigma \equiv N \cup T$  as the set of *symbols* in the grammar. Each production named  $p \in P$  has the form  $X_0^p \rightarrow X_1^p X_2^p \dots X_{n_p}^p$ , where  $X_0^p \in N, X_i^p \in \Sigma, 1 \leq i \leq n_p, n_p \geq 0$ . Define  $X^p \equiv \{X_i^p \mid 0 \leq i \leq n_p\}$ , the set of *symbol occurrences* for production  $p$ .

A classical attribute grammar with local attributes is a tuple  $(G, S, I, L, R)$  where  $G$  is a reduced<sup>1</sup> context-free grammar, and for each  $X \in \Sigma$ ,  $S(X)$  and  $I(X)$  are respectively the sets of *synthesized attributes* and *inherited attributes* defined for that symbol (with  $S(X) \cap I(X) = \emptyset$ ). Each terminal has no inherited attributes and a single synthesized attribute, *value*, (for  $X \in T, I(X) = \emptyset, S(X) = \{\text{value}\}$ ). Similarly the start symbol  $Z$  has no inherited attributes ( $I(Z) = \emptyset$ ). Define  $A(X) \equiv S(X) \cup I(X)$  as the total set of attributes for a symbol  $X$ . In a slight abuse of notation, we write  $S(X_i^p), I(X_i^p)$ , and  $A(X_i^p)$  to refer to the appropriate set of attributes for the nonterminal associated with the symbol occurrence  $X_i^p$ .

The example in Figure 3 has  $S(S) = \{x\}, S(A) = \{r, s\}, S(B) = \{t\}, I(S) = \{\}, I(A) = \{j\}, I(B) = \{i\}$  and uses no local attributes. The example is an abstraction of name resolution, where  $a$  represents a declaration and  $b$  represents a use, where  $c$  represents the absence of a use. The “use” (or lack of use) contributes information back to the declaration site and this information is “returned” as the result of the computation (the synthesized

<sup>1</sup>A *reduced* CFG has no useless nonterminals, ones that do not occur in any derivation.

attribute  $x$  of the root).

For each production  $p \in P$  of the form

$$X_0^p \rightarrow X_1^p X_2^p \dots X_{n_p}^p$$

$L^p$  is the set of local attributes. The set of *attribute occurrences* for  $p$  is the local attributes and the attributes of the nonterminal occurrences making up the production:

$$O^p \equiv L^p \cup \{X_i^p.a \mid a \in A(X_i^p)\}.$$

$R^p$  is a set of attribution rules relating attribute occurrences. Each rule  $r$  has the form

$$v_0 = g(v_1, \dots, v_k)$$

where  $k \geq 0$ , where for each  $0 \leq j \leq k$ ,  $v_j \in O^p$ , and where  $g$  is any (appropriately typed) strict function. Define  $DO(r) \equiv \{v_0\}$  as the *defined occurrence* of rule  $r$  (also known as the *output occurrence*), and  $UO(r) \equiv \{v_j \mid 0 < j \leq k\}$  as the *used occurrences* of rule  $r$  (also known as *input occurrences*).

In our example, many of the rules do not use explicit functions. A *copy rule* such as  $B.i = A.s$  includes the implicit application of the identity function.

The sets of attributes  $S(X)$ ,  $I(X)$  and  $L^p$  must all be finite for the attribute grammar to be *proper*. Otherwise the attribute grammar is *improper*. This article occasionally uses improper attribute grammars; results that are not true for improper attribute grammars will be so indicated. In particular, there is no attempt to define algorithms on improper attribute grammars.

The attribution rules must be well formed. The defined occurrence must be a local attribute, a synthesized attribute of the left-hand side of a production or an inherited attribute of the right-hand side:

$$DO(r) \subseteq DO^p$$

where  $DO^p$  is the set of all attribute occurrences for the production  $p$  that may be defined:

$$DO^p \equiv L^p \cup \{X_0^p.a \mid a \in S(X_0^p)\} \cup \{X_i^p.a \mid a \in I(X_i^p), 0 < i \leq n_p\}.$$

There must be precisely one definition for every defined occurrence:

$$\bigcup_{r \in R^p} DO(r) = DO^p$$

$$DO(r) \cap DO(r') = \emptyset, \quad r, r' \in R^p, r \neq r'.$$

For simplicity, the attribute grammar must be in Bochmann [1976] normal form. Only local attributes, the inherited attributes of the left-hand side and the synthesized attributes of the right-hand side may be used to compute an attribute value:

$$UO(r) \subseteq UO^p$$

where  $UO^p$  is the set of all the attribute occurrences for a production that may be used:

$$UO^p \equiv L^p \cup \{X_0^p.a \mid a \in I(X_0^p)\} \cup \{X_i^p.a \mid a \in S(X_i^p), 0 < i \leq n_p\}.$$

The example in Figure 3 has well-formed rules; indeed they are in Bochmann normal form. For instance, the rule  $B.t = f1(B.i)$  uses an inherited attribute of the left-hand-side and defines a synthesized attribute of the left-hand-side. The rule  $A.j = B.t$  shows

the definition of an inherited attribute of a right-hand-side symbol using a synthesized attributes of a different right-hand-side symbol.

An attribute grammar is *instantiated* for a given parse tree  $t$  describing the derivation of a terminal string from the start symbol of  $G$ . For the purposes of this article, all nodes in the tree have *identity*, perhaps by naming each node by the path from root, using some unspecified syntax for paths. For each non-terminal node using production  $p$  in the tree, the rules  $R^p$  are instantiated so that the equations are expressed in terms of *attribute instances*: attributes of the nodes of tree  $t$ . The well-formedness rules given above ensure that in the set of all attribute instance rules, every local attribute instance and every attribute instance  $n.a$  for a node  $n$  of nonterminal  $X$  (where  $a \in A(X)$ ) has exactly one definition. Let  $R(v)$  be the attribute instance rule for attribute instance  $v$ .

The semantics of the attribute grammar for the tree  $t$  is defined by solving the attribute instance rules simultaneously. Cyclic dependencies can be solved using a least fixed-point semantics [Farrow 1986; Rodeh and Sagiv 1999], but this article follows the classical definition of attribute grammars in requiring noncircularity. In some previous work, circularity is defined indirectly in terms of the compound dependency graph (explained below), but this article follows Knuth [1968] in defining circularity directly on the rules. This definition of circularity is also more easily translated to apply to remote attribute grammars, as described in Section 3.2.

In order to evaluate an attribute grammar, the attribute instances must be related by some strict total order  $<$  over the rule instances such that any attribute instance rule  $r$  that defines  $v_0$  ( $\{v_0\} = DO(r)$ ) must follow all the rules for the used attribute instances in the total order ( $v \in UO(r) \Rightarrow R(v) < R(v_0)$ ). Such a strict total order of the rule instances is called a *schedule*. Determining a schedule for any instantiation of a proper attribute grammar is straightforward if it exists. An attribute grammar is *noncircular* if a schedule exists for any tree; otherwise it is *circular*.

A noncircular proper attribute grammar can be *evaluated* for any tree by computing the attribute instances using the instantiated rules in the schedule. Much previous work on attribute grammars has concerned discovering families of schedules statically and using them for efficient evaluation of (proper) attribute grammars. Static scheduling uses an abstraction over the rules since the actual primitive functions used are irrelevant to the dependencies. Accordingly, we now turn to defining dependency graphs for attribute grammars.

A defined occurrence  $v \in DO^p$  is said to *depend* on a used occurrence  $v' \in UO^p$  (denoted  $v' \rightarrow v$ ) if the rule  $r \in R^p$  that defines  $v$  ( $\{v\} = DO(r)$ ) uses  $v'$  ( $v' \in UO(r)$ ). The *dependency graph* of a production is denoted  $D^p$ ; the vertices  $V_{D^p}$  are the attribute occurrences; the edges  $E_{D^p}$  are dependencies between attribute occurrences:

$$D^p \equiv (V_{D^p}, E_{D^p})$$

$$V_{D^p} \equiv O^p$$

$$E_{D^p} \equiv \{v' \rightarrow v \mid \exists r \in R^p, \{v\} = DO(r), v' \in UO(r)\}.$$

For any particular parse tree of the context-free grammar  $G$ , the dependency graphs for each production can be pieced together to construct a compound dependency graph for the whole tree. The close connection between the rules and the dependency graph is expressed in the following theorem:

$S \rightarrow A B$ $B.i = A.s$ $S.x = A.r$ $A \rightarrow a$ $\text{object } o$ $A.s = o$ $A.r = o.f$	$B \rightarrow b$ $\text{object } p$ $\text{local } l$ $l = B.i$ $l.f \sqsupseteq p$ $B \rightarrow c$ $(no\ rules)$
---	---

FIG. 4. A simple remote attribute grammar

**THEOREM 3.1.** *An attribute grammar is circular if and only if there exists a parse tree of the context-free grammar whose compound dependency graph has a cycle.*

**PROOF.** (Sketch)

The edges in the compound dependency graph represent precisely the constraints that the schedule must preserve. A cycle exists if and only if no total order exists that satisfies the constraints.  $\square$

Jazayeri, Ogden, and Rounds established that determining whether a (proper) attribute grammar is circular, is intrinsically exponential [Jazayeri et al. 1975].

**3.2. REMOTE ATTRIBUTE GRAMMARS.** A *remote attribute grammar* is a tuple  $(G, S, I, F, L, B, R)$  where  $G, S, I,$  and  $L$  are as with (proper) classical attribute grammars. The finite set  $F$  is a set of *fields* that objects have, and  $B$  is a finite set of named objects declared for each production. The attribute occurrences include the objects declared for a production as well as the occurrences of a classical attribute grammar:

$$O^p \equiv L^p \cup \{X_i.a \mid a \in A(X_i)\} \cup B^p.$$

The set of mandatory defined occurrences is the same as in a classical attribute grammar:

$$DO^p \equiv L^p \cup \{X_0^p.a \mid a \in S(X_0^p)\} \cup \{X_i^p.a \mid a \in I(X_i^p), 0 < i \leq n_p\}.$$

The set of used occurrences includes the objects.

$$UO^p \equiv L^p \cup \{X_0^p.a \mid a \in I(X_0^p)\} \cup \{X_i^p.a \mid a \in S(X_i^p), 0 < i \leq n_p\} \cup B^p.$$

In a remote attribute grammar, there are two forms of rules in addition to the classical form  $v_0 = g(v_1, \dots, v_n)$ . A rule  $r \in R^p$  may be of the form  $v = w.f$  (for  $f \in F$ ), which is called a *field read*, or it may be of the form  $w.f \sqsupseteq v$ , a *partial field write*. In the former case,  $v$  is defined and  $w$  is used:  $DO(r) \equiv \{v\}, UO(r) \equiv \{w\}$ . In the latter case, the rule uses both  $v$  and  $w$ ; it does not define anything local:  $DO(r) \equiv \{\}, UO(r) \equiv \{v, w\}$ .

The equal sign ( $=$ ) is not used in a partial field write because the rule does not define equality, instead it contributes something to the value of the field. The final value of the field is the combination of all the values that are written into it. This final value is used whenever the field is used in some other rule. These definitions avoid typing issues by assuming that all attributes and fields hold sets of objects. Thus  $w.f \sqsupseteq v$  means that for every object reference  $o$  in  $w$  that is a reference to an instance of an object declaration in the RAG, all the object references in  $v$  are added to field  $f$  of  $o$ . In a typed system, each

field may be assigned a default value and a combination function. Such extensions are discussed further in Section 3.4.

The primitive functions are permitted to pass through object instances and may also return “constant” objects that are not from the tree, but they are not allowed to return other object instances from the tree, which would make it impossible to track where the references come from or go to.

Figure 4 shows a simple example of a remote attribute grammar. It is designed to have a similar data-flow to the classical attribute grammar in Figure 3, although some of the flows occur through remote references. As in the classical case, the nonterminal A has a synthesized attribute  $s$ , which is copied into B’s inherited attribute  $i$ . Here, however, this value is an object created in the instance of A. Then if the B is an instance of the rule  $B \rightarrow b$ , the field of this object has added to it the object  $p$ . Instead of passing back a synthesized attribute from B into an inherited attribute of A, the object (or lack of object) is read into the resulting synthesized attribute  $r$  of A and then passed to the root in synthesized attribute  $x$ . Thus information flows from A to B through the tree and then, coming from B to A, through the remote reference. This flow would be seen as “node-wise circular” in Johnson’s non-local attribute grammars and thus could not be correctly implemented [Johnson 1983]. As mentioned previously, this example is an abstraction of the name resolution task in our first examples (Figures 1 and 2), in which the declaration is told whether or not it is used.

A well-formed remote attribute grammar must obey the same restrictions as a classical attribute grammar: no two rules may define the same attribute occurrence and the set of all defined occurrences from all the rules must be equal to  $DO^p$ . As with a classical attribute grammar, a remote attribute grammar is *instantiated* for a parse tree  $t$  by instantiating the rules for each node’s production. This process results in instances of the objects as well as the rules (the set of object instances is written  $B(t)$ ), and the rules define instances of fields of objects as well as attribute instances. One difference is that each field may have any number (zero or more) definitions. All definitions are combined to achieve the final value; in this untyped formalism, the partial field writes are seen as set constraints, and the smallest solution set is found for each field. As with classical attribute grammars, one can give a semantics that permits circular dependencies [Boyland 1996b; Magnusson and Hedin 2003; Sasaki and Sassa 2003], but this article restricts itself to noncircular remote attribute grammars.

A schedule for evaluation for a tree is acceptable if it ensures that attributes *and fields* are assigned before they are used. This restriction is complicated by the fact that the object(s) whose field is written may be determined by an attribute value. The acceptability of a schedule including such a rule may depend on which object(s) the attribute can have a reference to. For this purpose, it is important to know which arguments of primitive functions may transmit objects to the result. Some primitive functions transmit object references, but others simply use the references without transmitting them. For a function  $g$  with arity  $n$  and  $0 < i \leq n$ , the key  $L_{gi} \in \{0, u\}$  indicates the former and latter situations respectively. The 0 annotation is intended to be reminiscent of “no operation” whereas  $u$  stands for “uses” as opposed to “copies.” For example,  $L_{if1} = u, L_{if2} = L_{if3} = 0$ , since a reference to an object can be transmitted through the result of the “if,” but never through the condition.<sup>2</sup> One can always conservatively assume  $L_{gi} = 0$ ; in other words, this ap-

<sup>2</sup>Since the untyped system does not have Booleans, the empty set is considered to be “false” and all other sets “true.”

proximation may make the sets of reaching object references unnecessarily large, but it will never omit a reference to actual object that an attribute instance would evaluate to.

The analysis defines the sets  $B(v) \subseteq B(t)$  for each attribute instance  $v$  of an instantiated remote attribute grammar as sets of reaching object instance references. An object instance's reference is in the set if there is some path through the equations of the remote attribute grammar that can take a reference to that object instance to that attribute instance. Ignoring “constant” objects provided by primitive functions (which are not affected by remote writes), this set can be seen as an upper bound on the contents of attribute instance  $v$ .

The  $B(v)$  sets are defined as the least fixed-point solution of the following equations generated by the rule instances  $R(t)$ :

$$(v_0 = g(v_1, \dots, v_n)) \in R(t).$$

$$B(v_0) \supseteq B(v_i) \quad \text{when } L_{g_i} = 0$$

$$(v = w.f), (w'.f \sqsupseteq v') \in R(t).$$

$$B(v) \supseteq B(v') \quad \text{when } B(w) \cap B(w') \neq \emptyset$$

$$o \in B(t).$$

$$B(o) = \{o\}.$$

A schedule then for a remote attribute grammar is a strict total order  $<$  on the instantiated rules such that two conditions are met:

- (1) As with a classical schedule, every rule must be scheduled after any attribute it uses (objects are assumed immediately available):  $v \in UO(r) \Rightarrow R(v) < r \vee v \in B(t)$ .
- (2) Additionally, for two rules  $r_1 = (v = w.f), r_2 = (w'.f \sqsupseteq v') \in R(t)$  if these rules may be referring to the same object ( $B(w) \cap B(w') \neq \emptyset$ ), then the partial field write must precede the field read ( $r_2 < r_1$ ).

And as in the classical case, if a schedule exists for every parse tree  $t$ , then the remote attribute grammar is *noncircular*, otherwise *circular*.

A straightforward algorithm for evaluation of a remote attribute grammar is to perform this  $O(n^3)$  analysis and then use a topological sort to find a schedule (if one exists). However, the cubic-time worst case need not always apply. For instance, one can perform a coarser but faster analysis (such as requiring all writes of a field  $f$  for *any* object to occur before all reads of this field), or use a static scheduler (such as described later in this article).

Unlike the classical case, even if no schedule can be found, the rule instances may not actually involve a cyclic dependency: the sets  $B(v)$  approximate the actual objects involved and thus may overconstrain the schedule. Nevertheless, testing circularity of remote attribute grammars is undecidable. Circularity of remote attribute grammars is similar to interprocedural static analysis of recursive data structures which Reps [2000] showed undecidable. Undecidability is shown by a reduction from “Post’s Correspondence Problem.” [Hopcroft and Ullman 1979]:

An instance of *Post’s Correspondence Problem* (PCP) consists of two sequences of strings of the same length  $X = (x_1, \dots, x_n), Y = (y_1, \dots, y_n)$  over some alphabet  $\Sigma$ . The

problem asks whether there exists a finite nonempty sequence of integers  $i_1, \dots, i_m \in \{1, \dots, n\}$  such that

$$x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$$

For example, the following PCP over  $\Sigma = \{0, 1\}$  has solution 1, 2, 1:

$$\begin{aligned} x_1 &= 11 & y_1 &= 1 \\ x_2 &= 01 & y_2 &= 1011 \end{aligned}$$

The solution is verified by testing  $x_1 x_2 x_1 = 11 01 11 = 1 1011 1 = y_1 y_2 y_1$ .

**THEOREM 3.2** PCP [HOPCROFT AND ULLMAN 1979]. *Post's Correspondence Problem is undecidable.*

**THEOREM 3.3.** *Circularity of remote attribute grammars is undecidable.*

**PROOF.** This result is shown by reduction from PCP. Let  $X, Y$  be a Post's Correspondence Problem over  $\Sigma = \{0, 1\}$ . We construct a remote attribute grammar that is circular if and only if the PCP has a solution.

The intuition behind the construction is that there is one nonterminal production for each index in the PCP. The trees generated by the grammar are right-heavy with the sequence of integers in a PCP solution appearing in a preorder traversal of the tree. Each of the nonterminals representing  $(x_i, y_i)$  has one thread of dependencies that constructs objects nested according to  $x_i$  while going down the tree, and fetches objects according to  $y_i$  while going up the tree. It nests the object in field  $f_0$  or  $f_1$  while going down the tree and fetches through field  $f_0$  or  $f_1$  while going up the tree. With the addition of some glue productions, the result is achieved, that the reference passed down the tree is exactly the same as the reference that reappears on the way up if and only if the PCP sequence is indeed a solution. The construction is similar in spirit to the one used by Reps [2000] where the parse tree takes the place of the call tree for Reps, objects take the place of cons-cells,  $f_0$  takes the place of `car` and  $f_1$  takes the place of `cdr`.

The construction requires a set of three field  $\{f_0, f_1, g\}$ , a set of  $m$  local identifiers  $\{l_1, \dots, l_m\}$  and a set of  $m$  object identifiers  $\{o = o_1, \dots, o_m\}$  where  $m$  is the maximum length of any of the strings in the PCP. The object identifiers are reused in rules for different productions. Two of the nonterminals each have two attributes  $\{d, u\}$ . The construction uses  $d$  to refer to an inherited attribute carrying the reference to the object structure being built ("down") and  $u$  to refer to the synthesized attribute carrying the reference to the object being uncovered ("up").

Let  $G = (N, T, Z, P)$  be a context-free grammar where

$$\begin{aligned} N &= \{Z, V, W\}, \\ T &= \{T_1, \dots, T_n\}, \\ P &= \{V \rightarrow T_i W \mid 1 \leq i \leq n\} \cup \{Z \rightarrow V, W \rightarrow V, W \rightarrow \}. \end{aligned}$$

The nonterminal  $V$  derives a non-empty string of tokens, whereas  $W$  is possibly empty.

Let  $A = (G, S, I, F, L, B, R)$  be a remote attribute grammar where

$$\begin{aligned} S(V) &= S(W) = \{u\}, \\ I(V) &= I(W) = \{d\}, \\ F &= \{f_0, f_1, g\}, \end{aligned}$$

$$\begin{aligned}
L(Z \rightarrow V) &= \{l\}, \\
L(V \rightarrow T_i W) &= \{l_j \mid 1 \leq j \leq |y_i|\}, \\
L(W \rightarrow V) = L(W \rightarrow) &= \{\}, \\
B(Z \rightarrow V) &= \{o\}, \\
B(V \rightarrow T_i W) &= \{o_j \mid 1 \leq j \leq |x_i|\}, \\
B(W \rightarrow V) = B(W \rightarrow) &= \{\},
\end{aligned}$$

and  $R$  is defined as follows, where the rules for each production are given in turn with some commentary to explain the construction:

$$\begin{aligned}
Z \rightarrow V \\
o . g &\sqsupseteq l \\
V . d &= o \\
l &= V . u . g
\end{aligned}$$

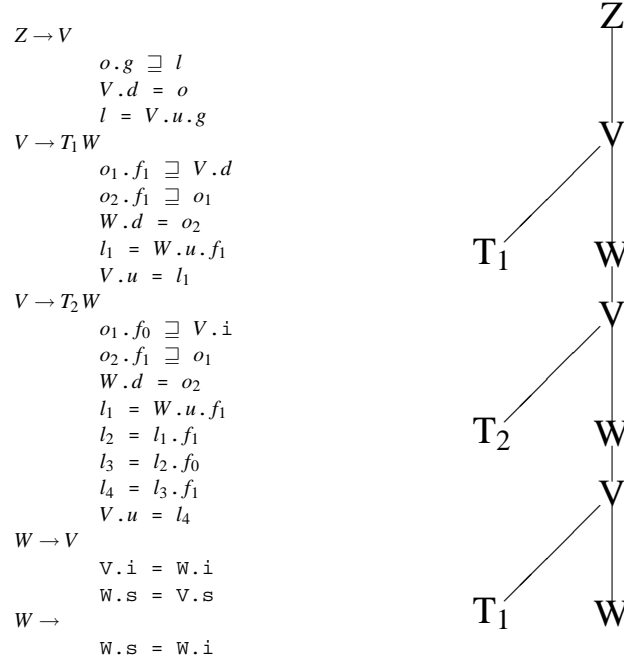
This production (which always occurs at the root) declares an object with a single field  $g$  initialized to the value of the local  $l$ , which comes from performing a remote read on the object referenced by  $V . u$ . If this object is the same as  $o$ , there is a cycle. Otherwise not. The reason why a cycle is possible is that a reference to  $o$  is sent down in the  $V . d$  attribute.

Each of the  $V \rightarrow T_i W$  productions has the rules:

$$\begin{aligned}
V \rightarrow T_i W \\
o_1 . f_{x_{i1}} &\sqsupseteq V . d \\
&\vdots \\
o_j . f_{x_{ij}} &\sqsupseteq o_{j-1} \\
&\vdots \\
W . d &= o_{|x_i|} \\
l_1 &= W . u . f_{y_{i1}}^R \\
&\vdots \\
l_j &= l_{j-1} . f_{y_{ij}}^R \\
&\vdots \\
V . u &= l_{|y_i^R|}
\end{aligned}$$

Here  $x_{ij}$  is the  $j$ 'th bit of  $x_i$  and  $y_{ij}^R$  is the  $j$ 'th bit of the *reverse* of string  $y_i$ . These rules perform the meat of the construction. Here  $W . d$  has a reference to an object where the input object reference  $V . d$  is buried using exactly the fields defined by the string  $x_i$ . Similarly, the reference  $V . u$  dereferences the input reference  $W . u$  by the *reverse* of the string of  $y_i$ . The reversal is necessary since the rules apply in the opposite direction (up rather than down, in rather than out). An important feature of the construction is that it does not make use of remote field writes; all fields are assigned only in the production where the object is declared. Indeed the only writes are of the form  $o_i . f_j \sqsupseteq \dots$ , and thus this proof shows that undecidability is due to the presence of remote reads alone.

The  $W \rightarrow V$  production is used when the construction continues (there are more tokens to parse):


 FIG. 5. RAG constructed for PCP  $X = (11, 01), Y = (1, 1011)$  and the cycle-demonstrating tree

$$\begin{aligned}
 W &\rightarrow V \\
 & \quad V.d = W.d \\
 & \quad W.u = V.u
 \end{aligned}$$

The references are passed through unchanged.

At the bottom of the tree, there is an empty  $W$ :

$$\begin{aligned}
 W &\rightarrow \\
 & \quad W.s = W.i
 \end{aligned}$$

The reference that comes down is passed back up. The reason  $W$  and  $V$  are distinguished (and thus have the extra glue productions) is without the distinction, one could form a tree with just the top and bottom productions. Such a tree would *always* exhibit a cycle and thus would be useless. It corresponds to the empty sequence as a trivial “solution” to the PCP.

Now we claim that  $A$  is circular if and only if the PCP has a solution. Let  $i_1, \dots, i_m \in \{1..n\}$  be a non-empty sequence of integers that may or may not be a solution to this PCP. Let  $\alpha = x_{i_1} \dots x_{i_m}, \alpha' = y_{i_1} \dots y_{i_m}$ . This sequence is a solution if and only if  $\alpha = \alpha'$ .

Let  $t$  be the tree for the yield  $T_{i_1} \dots T_{i_m}$ . The grammar is right-linear and thus the tree is linear. A sample tree may be seen on the right of Figure 5. In general a tree has the form:

$$\begin{array}{ccccccc}
 Z & \rightarrow & V & \rightarrow & W & \rightarrow & V & \rightarrow & \dots & \rightarrow & V & \rightarrow & W \\
 & & & & \searrow & & & & & & \searrow & & \\
 & & & & & & T_{i_1} & & & & & & T_{i_m}
 \end{array}$$

The instantiated rules take the following form (where the instances of  $V$  and  $W$  are num-

bered from the root in a superscript):

$$\begin{array}{ccc}
o_1^1 \cdot f_{x_{i_1}} \supseteq V^1 \cdot d, & & o_1^m \cdot f_{x_{i_m}} \supseteq V^m \cdot d, \\
\cdots, & & \cdots, \\
o_j^1 \cdot f_{x_{i_j}} \supseteq o_{j-1}^1, & & o_j^m \cdot f_{x_{i_j}} \supseteq o_{j-1}^m, \\
\cdots, & & \cdots, \\
W^1 \cdot d = o_{|x_{i_1}|}^1, & & W^m \cdot d = o_{|x_{i_m}|}^m, \\
o \cdot g \supseteq l & l_1^1 = W^1 \cdot u \cdot f_{y_{i_1}}^R, & \vdots & l_1^m = W^m \cdot u \cdot f_{y_{i_m}}^R, \\
V^1 \cdot d = o & \cdots, & & \cdots, \\
l = V^1 \cdot u \cdot g & l_j^1 = l_{j-1}^1 \cdot f_{y_{i_j}}^R, & & l_j^m = l_{j-1}^m \cdot f_{y_{i_j}}^R, \\
& \cdots, & & \cdots, \\
& V^1 \cdot u = l_{|y_{i_1}|}^1, & & V^m \cdot u = l_{|y_{i_m}|}^m, \\
& & & \\
V^2 \cdot d = W^1 \cdot d & & & W^m \cdot u = W^m \cdot d \\
W^1 \cdot u = V^2 \cdot u & & & 
\end{array}$$

Many of these rules are copy rules, such as the following:

$$W^k \cdot d = o_{|x_{i_k}|}^k, \quad V^k \cdot u = l_{|y_{i_k}|}^k, \quad V^{k+1} \cdot d = W^k \cdot d, \quad W^k \cdot u = V^{k+1} \cdot u$$

The rules for  $B(v)$  (on page 14) are such that these copy rules can be ignored. If one factors out all the copy rules and ignores where the various attribute instances came from, one has a set of rules of following form:

$$\begin{array}{ccc}
o^1 \cdot f_{\alpha_1} \supseteq o^0 & & l^1 = l^2 \cdot f_{\alpha'_1} \\
o^2 \cdot f_{\alpha_2} \supseteq o^1 & & l^2 = l^3 \cdot f_{\alpha'_2} \\
\vdots & & \vdots \\
o^{|\alpha|} \cdot f_{\alpha_{|\alpha|}} \supseteq o^{|\alpha|-1} & & l^{|\alpha|} = o^{|\alpha|} \cdot f_{\alpha'_{|\alpha|}}
\end{array}$$

Now if the sequence is a solution, then  $\alpha = \alpha'$  and thus the rules for  $B$  can be used on the equations starting with the bottom to achieve  $B(l^{|\alpha|}) = \{o^{|\alpha|-1}\}, \dots, B(l^1) = \{o^0\}$ . Suppose one were to try to make a schedule. In particular one would need to order the rules:  $o^0 \cdot g \supseteq l^0$  and  $l^0 = l^1 \cdot g$ . This order must meet both conditions:

- (1)  $l^0 = l^1 \cdot g < o^0 \cdot g \supseteq l^0$  because the left rule defines  $l_0$ , which is used in the right rule; and
- (2)  $o^0 \cdot g \supseteq l^0 < l^0 = l^1 \cdot g$  because the left rule writes a field that the other may read, since  $B(l^1) = \{o^0\}$ .

These conditions are contradictory. No schedule can be found and thus the constructed RAG is circular.

Suppose on the other hand, there is no solution to the PCP. Then consider any tree generated by the grammar. This tree must be of the form shown above. Let  $i_1, \dots, i_m$  be the sequence. Since the PCP has no solution,  $\alpha \neq \alpha'$ . Next consider four cases:

- Suppose  $\alpha$  and  $\alpha'$  differ in their last bits:  $\alpha_{|\alpha|} \neq \alpha'_{|\alpha'|}$ . Then  $f_{\alpha_{|\alpha|}} \neq f_{\alpha'_{|\alpha'|}}$ , and since there is no other partial field write for a field of  $o^{|\alpha|}$ , the reaching object set for  $l^{|\alpha|}$  must

be empty, and thus the reaching object sets for all previous  $l$ 's must also be empty and thus  $B(l^1)$  is also empty.

- If  $\alpha$  and  $\alpha'$  are the same in their last bits but differ somewhere earlier, the same argument applies starting from the point where they differ, and thus  $B(l^1) = \emptyset$ .
- If  $\alpha$  is a suffix of  $\alpha'$  ( $\beta\alpha = \alpha'$ , where  $|\beta| \geq 1$ ), then  $B(l_{|\beta|+1}) = \{o^0\}$ . Now since there is no  $f_0$  or  $f_1$  field assigned to  $o^0$ , then  $B(l_{|\beta|}) = \emptyset$ , and no objects reach any previous  $l$  either and thus  $B(l^1) = \emptyset$ .
- If, rather,  $\alpha'$  is a suffix of  $\alpha$  ( $\alpha = \beta'\alpha'$  where  $|\beta'| \geq 1$ ), then  $B(l_1) = \{o^{|\beta'|}\}$ .

In all cases,  $o^0 \notin B(l^0)$ , thus the rules can be ordered as follows:

$$o^1 \cdot f_{\alpha_1} \sqsupseteq o^0 < o^2 \cdot f_{\alpha_2} \sqsupseteq o^1 < \dots < o^{|\alpha|} \cdot f_{\alpha_{|\alpha|}} \sqsupseteq o^{|\alpha|-1} < \\ l^{|\alpha|} = o^{|\alpha|} \cdot f_{\alpha'_{|\alpha|}} < \dots < l^2 = l^3 \cdot f_{\alpha'_2} < l^1 = l^2 \cdot f_{\alpha'_1} < l^0 = l^1 \cdot g < o^0 \cdot g = l^0.$$

Of course, the actual rules must be scheduled, not a list with the copy rules removed. This task is accomplished by putting the copy rules in the schedule just before the point where they are used.

Since a schedule is thus always possible, the constructed RAG is noncircular. Therefore the constructed RAG is circular if and only if the PCP has a solution, and thus circularity of remote attribute grammars is undecidable.  $\square$

In the construction used in the proof, all of the writes of fields were local to the point where the objects were created. Thus the undecidability is due merely to the use of remote reads, not due to remote writes. Farrow conjectured the undecidability of circularity of remote attribute grammars without collection fields [Farrow 1990]; this article includes the first proof of the fact.

A normal dependency graph between definitions and uses does not capture all the dependencies in a remote attribute grammar; none of the dependencies induced through read and writes of fields are represented. The following portion of the article defines a kind of dependency graph with labeled edges that *does* capture the dependencies, and it also defines what it means for there to be a cyclic dependency. It shows that circularity of an instance of an RAG is a “context-free reachability” problem [Reps 1998].

**3.3. FIELD OPERATION DEPENDENCY GRAPH.** The field operation dependency graph is a dependency graph where edges have labels that, in essence, report what operation it does to the object as it traverses the edge. To help introduce this construct slowly, first assume that there are no remote writes (as in the construction used to prove undecidability). For now, the construction uses unlabeled edges to refer to classical attribute grammar dependencies, but two kinds of labeled edges are introduced:

$v \xrightarrow{f} o$ . This edge means that the  $f$  field of object  $o$  is defined to be the value  $v$ . (This edge is used only if there are not any collections.)

$v \xrightarrow{f^{-1}} w$ . This edge means that  $w$  depends on the field  $f$  of any object references by  $v$ .

Figure 6 shows the field operation dependency graph for the example tree and RAG from Figure 5. There is an obvious cycle, but is it one that renders the RAG unschedulable? To determine whether it is what we call a “balanced” cycle, one sees whether the  $f$  and



$w$ . This kind of edge corresponds to what appears in a classical dependency graph. For instance, the rule  $w = v$  induces an edge with this label.

$v \xrightarrow{u} w$ . The dependency here is that the value of  $v$  is *used* to compute the value of  $w$  but no references from  $v$  flow into  $w$  by this route. In some classical attribute grammars, there is a concept of “control” dependencies. Control dependencies are seen as a kind of “use” dependency. The rule  $w = \text{if}(v, \dots, \dots)$  induces an edge of this form.

$v \xrightarrow{f^{-1}} w$ . As explained above, this means that  $w$  reads the  $f$  field on any object reference in  $v$ . Such an edge comes from a rule  $w = v.f$ .

$v \xrightarrow{f} w$ . This edge handles *partial* writes: the value of  $v$  is used to partially specify the field  $f$  of an object referenced by  $w$ . This edge comes from a rule of the form  $w.f \sqsupseteq v$ .

$w \xrightarrow{uf} w$ . In order to perform a partial write  $w.f \sqsupseteq v$ , we need to have the references themselves. In other words, one cannot schedule the partial write until after the attribute that carries the references to the objects whose field is being written. The  $u$  part of this label indicates that the references in  $w$  are *used* to perform the field write, but are not (necessarily) transmitted.

Now these partial field values must travel back to the place where the object is declared so that all the partial specifications can be collected in one place. The next kind of edge does the processing:

$o \xrightarrow{f^{-1}f} o$ . This edge label has conceptually two parts (although formally it is a single token). The first part fetches all the partial writes for field  $f$  that were sent back to the object and the second part uses the result to assign the full (final) value of the  $f$  field.

$o \xrightarrow{uf} o$ . Along with the partial writes, implementation needs to use the object reference itself as a key to collect the appropriate partial fields (the one for this object, not other objects).

In order to get the partial writes back to the object, one traverses edges that are in the opposite direction of the “normal” attribute flow. Thus there are edges such as the following:

$v \xrightarrow{\bar{0}} w$ . This edge is induced whenever a reference may flow from  $w$  to  $v$ ; partial writes flow from  $v$  to  $w$ . Whenever there is a rule  $v = g(\dots, w, \dots)$  (with  $w$  the  $i$ th actual parameter) and  $L_{gi} = 0$  (for example when  $g$  is the identity function and  $i=1$ ), the construction not only adds the edge  $w \xrightarrow{0} v$  (as seen above) but also  $v \xrightarrow{\bar{0}} w$ .

$v \xrightarrow{\bar{u}} w$ . For simplicity, the construction also defines edges labeled  $\bar{u}$  although no information flows along them. This label does not occur in the CFG of “balanced strings” of labels.

Now this “backward” flow of partial writes travels not only through copy rules, but also through the fields of other objects:

$v \xrightarrow{\bar{f}} w$ . This edge transmits partial writes (of other fields) from  $v$  back along the path that defined the full definition of field  $f$ . It is induced by a rule of the form  $v = w.f$ .

$o \xrightarrow{\bar{f}^{-1}\bar{f}} o$ . When such values reach back to the object, they are collected and then broadcast out to all the partial writes, following the normal attribute flow direction.

$v \xrightarrow{\hat{f}^{-1}} w$ . When these partial writes of *other* fields reach back to a partial write of  $f$ , they are removed, and sent in backward flow back through  $w$ . This edge is generated from a rule  $v.f \sqsupseteq w$ .

The field operation dependency graph can now be defined.

The *field operation dependency graph* ( $D_F^p$ ) for a production  $p$  in a remote attribute grammar has the following form:

$$\begin{aligned} D_F^p &\equiv (V_{D_F^p}, E_{D_F^p}) \\ V_{D_F^p} &\equiv O^p \\ E_{D_F^p} &\equiv \{v_i \xrightarrow{L_{g_i}} v_0, v_0 \xrightarrow{\bar{L}_{g_i}} v_i \mid (v_0 = g(v_1, \dots, v_n)) \in R^p\} \cup \\ &\quad \{v \xrightarrow{\hat{f}} w, w \xrightarrow{uf} w, w \xrightarrow{\hat{f}^{-1}} v \mid (w.f \sqsupseteq v) \in R^p\} \cup \\ &\quad \{w \xrightarrow{f^{-1}} v, w \xrightarrow{u} v, v \xrightarrow{\bar{f}} w \mid (v = w.f) \in R^p\} \cup \\ &\quad \{o \xrightarrow{\hat{f}^{-1}f} o, o \xrightarrow{uf} o, o \xrightarrow{\bar{f}^{-1}\hat{f}} o \mid f \in F, o \in B^p\} \end{aligned}$$

As usual, a *compound field operation dependency graph* for a parse tree is formed by piecing together a field operation dependency graph for each production that occurs in the tree.

Figure 7 gives an example field operation dependency graph using rules from a single production. A path from  $a$  to  $b$  is outlined in the figure. A path in a (compound) field operation dependency graph is *balanced* if the strings formed by concatenating the edge labels can be derived from  $S$  in the following context-free grammar  $G_L$  ( $\lambda$  refers to the empty string):

$$\begin{aligned} S &::= SS \mid P \mid u \mid u\hat{f}_1\bar{P}\hat{f}_1^{-1}f_1P\hat{f}_1^{-1} \mid \dots \mid u\hat{f}_n\bar{P}\hat{f}_n^{-1}f_nP\hat{f}_n^{-1} \mid u\hat{f}_1P\hat{f}_1^{-1} \mid \dots \mid u\hat{f}_nP\hat{f}_n^{-1} \\ P &::= PP \mid \lambda \mid 0 \mid \hat{f}_1\bar{P}\hat{f}_1^{-1}f_1P\hat{f}_1^{-1} \mid \dots \mid \hat{f}_n\bar{P}\hat{f}_n^{-1}f_nP\hat{f}_n^{-1} \\ \bar{P} &::= \bar{P}\bar{P} \mid \lambda \mid \bar{0} \mid \bar{f}_1\bar{P}\bar{f}_1^{-1}\bar{f}_1P\bar{f}_1^{-1} \mid \dots \mid \bar{f}_n\bar{P}\bar{f}_n^{-1}\bar{f}_nP\bar{f}_n^{-1} \end{aligned}$$

(This grammar is structured on complete edge labels: the single edge label  $u\hat{f}$  is to be distinguished from the sequence  $u\hat{f}$ . This distinction makes no difference in the generated strings, but it permits us to use induction over derivations.) This grammar only allows  $u$ ,  $uf$  and  $u\hat{f}$  edges to be traversed between balanced segments; in other words, these dependencies are only valid between actual attribute occurrences/instances, not the fields of the objects carried by them. The edge label  $\bar{u}$  cannot be used anywhere. Figure 7 includes a simple proof that the outlined path from  $a$  to  $b$  is a balanced path.

A *balanced cycle* is a non-empty balanced path that starts and ends at the same node. Determining whether a balanced cycle exists is thus a ‘‘context-free reachability’’ problem for which Reps [1998] has given a  $O(n^3)$  algorithm. In the material that follows, the notation ‘‘ $v \xrightarrow{\gamma} w$ ’’ means that a path exists from  $v$  to  $w$  such that the labels along the edge are within the yield of the string of grammar symbols  $\gamma$ . For example  $v \xrightarrow{S} w$  means that a balanced path exists from  $v$  to  $w$ .

As with classical attribute grammars, the field operation dependency graph fully characterizes circularity. The proof of this result uses two technical lemmas:

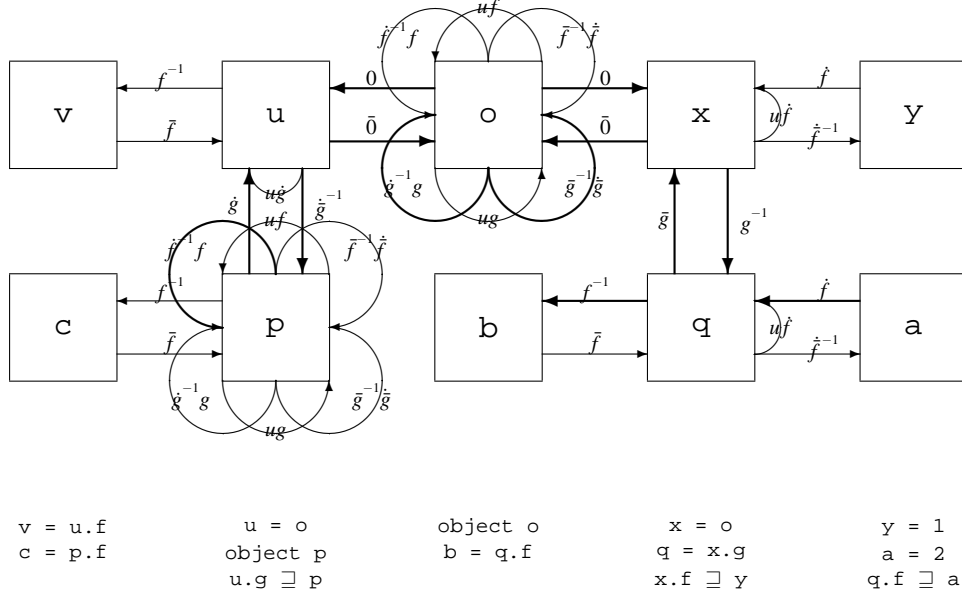


FIG. 7. Example field operation dependency graph with outlined path from a to b

LEMMA 3.4. *Given the instantiation of a remote attribute grammar  $A$  for a tree  $t$  and given its compound field operation dependency graph, then for every attribute instance  $v$  and every object instance  $o$ , the following statements are equivalent:*

- (1)  $o \in B(v)$
- (2)  $o \xrightarrow{P} v$
- (3)  $v \xrightarrow{\bar{P}} o$

PROOF. The result is proved in two parts:

(1) $\Rightarrow$ (2) and (1) $\Rightarrow$ (3). We prove the result by induction over the derivation of  $o \in B(v)$ . If  $v = o$ , then we are done since  $o \xrightarrow{\lambda} o$  and thus  $o \xrightarrow{P} o$  and  $o \xrightarrow{\bar{P}} o$ . If instead the result came from the rule  $v = g(\dots, v_i, \dots)$ , then  $L_{gi} = 0$  must hold, and also by induction  $o \xrightarrow{P} v_i$  and  $v_i \xrightarrow{\bar{P}} o$ . From this, it is easily seen that  $o \xrightarrow{P0} v$  and  $v \xrightarrow{\bar{0}\bar{P}} o$  from which the result follows using  $P \Rightarrow PP \Rightarrow P0$  and  $\bar{P} \Rightarrow \bar{P}\bar{P} \Rightarrow \bar{0}\bar{P}$ . The last case to consider is the case with a field write and read:  $v = w.f, w'.f \sqsupseteq v'$  where  $B(w) \cap B(w') \neq \emptyset$ . Let  $o'$  be some object in this non-empty intersection. By induction, then  $o' \xrightarrow{P} w, o' \xrightarrow{P} w', w \xrightarrow{\bar{P}} o, w' \xrightarrow{\bar{P}} o$ .

(2) $\Rightarrow$ (1) and (3) $\Rightarrow$ (1). We prove this by proving something stronger: that  $v' \xrightarrow{P} v$  or  $v \xrightarrow{\bar{P}} v'$  implies  $B(v) \supseteq B(v')$ . The result follows by setting  $v' = o$  since  $B(o) = \{o\}$ . The

stronger result is proved by induction over the grammar derivation of the paths between  $v'$  and  $v$ . Thus we assume that the result holds for all shorter derivations of  $P$  or  $\bar{P}$ . Consider first the case where  $v' \xrightarrow{P} v$ . Looking at the derivation of  $P$ :

$P \Rightarrow PP$ . There must be a  $w$  such that  $v' \xrightarrow{P} w \xrightarrow{P} v$ . Then by induction  $B(v) \supseteq B(w) \supseteq B(v')$  and we are done.

$P \Rightarrow \lambda$ . In this case  $v = v'$  and the result is immediate.

$P \Rightarrow 0$ . This edge must come from a rule instance  $v = g(\dots, v_i, \dots)$  where  $v' = v_i$  and  $L_{gi} = 0$ . This lets us immediately determine  $B(v) \supseteq B(v')$ .

$P \Rightarrow \dot{f}\bar{P}\dot{f}^{-1}fP\dot{f}^{-1}$ . This string is possible only when there exist rule instances  $w.f \sqsupseteq v'$  and  $v = w'.f$  and an object instance  $o'$  where  $v' \xrightarrow{\dot{f}} w \xrightarrow{\bar{P}} o' \xrightarrow{\dot{f}^{-1}f} o' \xrightarrow{P} w' \xrightarrow{\dot{f}^{-1}} v$ . Then by induction  $o' \in B(w)$  and  $o' \in B(w')$  and thus  $B(w) \cap B(w') \neq \emptyset$  and therefore  $B(v) \supseteq B(v')$ .

The case for  $v \xrightarrow{\bar{P}} v'$  is completely analogous.

□

LEMMA 3.5. *Given the instantiation of a remote attribute grammar  $A$  for a tree  $t$  and two attribute instances  $v$  and  $v'$ , then a constraint on their defining rules  $R(v) < R(v')$  is induced if and only if a non-empty path exists  $v \xrightarrow{S} v'$ .*

PROOF.  $\Rightarrow$ . If the constraint on rules  $R(v) < R(v')$  is induced, it must either be the result of transitivity, the rule that every variable's defining rule must come before any of its using rules, or the rule that writes of fields must come before the reads. The last case, however, does not apply in this situation because the write of a field is not the defining rule for any attribute and thus cannot be  $R(v)$  for any  $v$ . In the "middle" case  $v \in UO(r')$  where  $r' = R(v')$ . There are two possibilities for  $r'$ :

$v' = v.f$ . In this case,  $v \xrightarrow{u} v'$  and thus  $v \xrightarrow{S} v'$  and we are done.

$v' = g(\dots, v_i = v, \dots)$ . In this case,  $v \xrightarrow{u} v'$  (in which case we are done as in the previous case) or  $v \xrightarrow{0} v'$ , in which case  $v \xrightarrow{P} v'$  and thus  $v \xrightarrow{S} v'$ .

Thus the only remaining case is transitivity:  $R(v) < r < R(v')$ . Without loss of generality, let  $r < R(v')$  be a base (non-transitive) case. If  $r' = R(w)$  for some  $w$ , then by induction  $v \xrightarrow{S} w \xrightarrow{S} v'$  and thus  $v \xrightarrow{SS} v'$  and we are done (since  $S \Rightarrow SS$ ). Otherwise,  $r$  must be a field write  $w.f \sqsupseteq v''$  and  $r' = R(v')$  must be a field read of the same field  $v' = w'.f$  where  $B(w) \cap B(w') \neq \emptyset$ . Let  $o$  be some object in this non-empty intersection. Then from the definition of the field operation dependency graph we achieve  $v'' \xrightarrow{\dot{f}} w$ ,  $w \xrightarrow{uf} w$  and  $w' \xrightarrow{\dot{f}^{-1}} v'$ . Using Lemma 3.4, one can determine  $w \xrightarrow{\bar{P}} o$  and  $o \xrightarrow{P} w'$  and from the fact  $o$  is an object, we get  $o \xrightarrow{\dot{f}^{-1}f} o$ . Putting all these parts together, we achieve the two paths:

$$\begin{array}{c} v'' \xrightarrow{\dot{f}} w \xrightarrow{\bar{P}} o \xrightarrow{\dot{f}^{-1}f} o \xrightarrow{P} w' \xrightarrow{\dot{f}^{-1}} v' \\ w \xrightarrow{uf} w \xrightarrow{\bar{P}} o \xrightarrow{\dot{f}^{-1}f} o \xrightarrow{P} w' \xrightarrow{\dot{f}^{-1}} v' \end{array}$$

and thus  $v'' \xrightarrow{S} v'$  and  $w \xrightarrow{S} v'$ .

Now the constraint  $R(v) < r$  must come from somewhere and must come either through  $R(v) \leq R(w)$  or  $R(v) \leq R(v'')$ . In the first case, either  $v = w$  and thus we are done, or else

$v \xrightarrow{S} w$  by induction and thus  $v \xrightarrow{SS} v'$  and thus  $v \xrightarrow{S} v'$ . The second case similarly yields  $v = v''$  or  $v \xrightarrow{S} v'' \xrightarrow{S} v'$  and we are done.

$\Leftarrow$ . We prove the stronger result that the result holds for a path labelled with a non-empty yield of  $S$  or  $P$  and prove the result by induction over the derivation:

$S \Rightarrow SS$ . Without loss of generality, neither part  $S$  has an empty yield. Then the result holds by induction and transitivity.

$S \Rightarrow P$ . The result holds by induction.

$S \Rightarrow u$ . A single edge  $u$  occurs only for  $R(v') = r' = v' = g(\dots, v_i = v, \dots)$  where  $L_{gi} = u$  in which case  $v \in UO(r')$  and thus  $R(v) < r' = R(v')$  and we are done.

$S \Rightarrow uf\bar{P}\bar{f}^{-1}fPf^{-1}$ . In this case, there must be an object  $o$  and a rule instance  $r = v.f \sqsupseteq v''$  where  $v \xrightarrow{\bar{P}} o \xrightarrow{P} w'$  and a second rule instance  $r' = v' = w'.f$ . By Lemma 3.4,  $o \in B(v) \cap B(w')$  and thus  $r < r' = R(v')$ . But since  $v \in UO(r)$  it must be the case that  $R(v) < r$  and thus we have our result by transitivity.

$P \Rightarrow PP$ . The result holds by induction and transitivity. (Again, the empty yield case can be excluded.)

$P \Rightarrow \lambda$ . This case cannot occur.

$P \Rightarrow 0$ . This case is a minor variation on the case for  $S \Rightarrow u$ .

$P \Rightarrow \dot{f}\bar{P}\bar{f}^{-1}fPf^{-1}$ . In this case, there must be an object  $o$  and a rule instance  $r = w.f \sqsupseteq v$  where  $w \xrightarrow{\bar{P}} o \xrightarrow{P} w'$  and a second rule instance  $r' = v' = w'.f$ . By Lemma 3.4,  $o \in B(w) \cap B(w')$  and thus  $r < r' = R(v')$ . But since  $v \in UO(r)$  then  $R(v) < r$  and thus we have our result by transitivity.

□

Now follows a version of the circularity theorem for field operation dependency graphs:

**THEOREM 3.6.** *A remote attribute grammar is circular if and only if there exists a parse tree  $t$  for which the compound field operation dependency graph has a balanced cycle.*

**PROOF.** Suppose that such a balanced cycle exists:  $v \xrightarrow{S} v$ . If  $v$  were an object instance, the last edge  $w \xrightarrow{l} v$  has an object as a sink. Since objects are used (but never defined), inspection of the generated edges shows that the label must come from the set  $\{\bar{0}, \bar{u}, \bar{f}, u\dot{f}, \dot{f}^{-1}, \bar{f}, \bar{f}^{-1}f, uf, \bar{f}^{-1}\dot{f} \mid f \in F\}$ . The label must also be a possible “ending” for  $S$ , and thus (by inspection of the grammar) must come from the set  $\{0, u, f^{-1}\}$ . No label satisfies both requirements, and thus  $v$  must *not* be an object instance, and must rather be an attribute instance. Lemma 3.5 then tells us that the constraint on the rule defining  $v$  has the form:  $R(v) < R(v)$  which is unsatisfiable by any schedule.

On the other hand, if the remote attribute grammar is circular, there must be a (perhaps transitive) constraint  $R(v) < R(v)$ , which by Lemma 3.5 tells us that  $v \xrightarrow{S} v$  which is a balanced cycle. □

In classical attribute grammars, the concept of dependency graphs is useful for defining “evaluation classes” which statically approximate the shape of the compound dependency graph. The field operation dependency graph does not lend itself to the same set of approximations because of the difficulty of providing labels on summary dependency edges. One

way forward would be to approximate the labels on the summary edges with regular expressions. However, this article takes a different tack. Section 4 describes a construction in which the dependencies of a remote attribute grammar are expressed in an improper classical attribute grammar. This construction together with projection operations that approximate the improper attribute grammars in proper form enables practical implementation. First, however, there follows a discussion of practical remote attribute grammars.

3.4. EXTENSIONS. The remote attribute grammars described in this section lack some of the convenient features used in the example. Now we describe how some simple extensions can be encoded in remote attribute grammars. The remainder of this article, however, will operate with the basic unextended remote attribute grammars.

The form of rules in the definition is very restrictive. In a practical system, one permits “expressions” wherever an attribute is used, so that for instance  $w.f \sqsupseteq v$  is extended to  $e_1.f \sqsupseteq e_2$ . An expression can be an attribute occurrence  $v$ , a primitive function call  $g(e_1, \dots, e_n)$  or a field read  $e.f$ . This extension can be easily accomplished by generating new local attributes.

The example remote attribute grammar in Figure 2 uses “global collections.” A global collection may be seen as a field of a special object created at the root and then passed down to every node of the tree through a new inherited attribute for every nonterminal. Any node may therefore add something to the global collection or request its final value. Global collections are a convenient place to place error messages and information about the global scope (if the attribute grammar is checking static semantics).

Not all fields need to be remotely written. An attribute grammar writer may indicate that some field may only be written using the object directly. If this distinction is made, the remaining fields, which may be remotely written, are called *collection fields*.

As mentioned previously, the formal system assumes that the value of each field or attribute is always a bag of objects, possibly empty. A practical system has types that are used to type fields, attributes and even objects. A straight-forward type-checker ensures that types are used properly. Collection fields must either be given types which have an appropriately defined collection operations, or else give the initial value and combination function. If the collection field has type  $S$  and the remote writes each add values of type  $T$ , then the initial value has type  $S$  and the combination function has type  $S \times T \rightarrow S$ . A combination function  $f$  must satisfy the property that  $f(f(v,x),y) = f(f(v,y),x)$  so that the order that values are added to the collection does not affect the outcome. If  $S = T$ , it is sufficient that  $f$  be commutative and associative. Section 6 discusses combination functions in the context of practical implementation.

Unlike our earlier work [Boyland 1996b], the source tree nodes cannot be passed through the attribute system, only newly created objects. The ability, however, can be easily simulated by having the system automatically create objects that mirror each tree node and store the node’s attributes as fields. If the ability to traverse the tree structure is also desired, this can be provided through additional fields. Furthermore, “collection attributes” on nodes can be simulated by writing the value of the attribute from a (collection) field on the mirror object. Thus all these extensions can be expressed in the base semantics before analysis. At implementation time, the mirror objects can be ignored and all work done on the actual tree nodes.

The example in Figure 2 uses conditional attribute grammars [Boyland 1996a]. Conditional attribute grammars complicate scheduling but orthogonally to the issues raised by

remote attribute grammars. Our implementation handles conditional attribute grammars, but given the separability of the issues, this article will not discuss it further.

The example also uses a semantic function that accesses fields of objects. Thus the function has additional dependencies that cannot be determined simply by examining the actual parameters. In fact the definition of circularity of remote attribute grammars is incorrect if a semantic function can access fields—not only does it end up with possibly incomplete  $B$  sets, but a field read in the function body may not correctly reflect all writes to that field. Thus, the function body must be treated as a set of (conditional) rules for the unique production of a new nonterminal. Then the function call is treated as a child of the production (or function!) whose rules include it. This technique (which handles recursion in functions in the same manner as recursion in grammars) was proposed by Parigot et al. [1996] in their “dynamic attribute grammars.” When the function is implemented, as it happens, all the real work happens in the first “visit” and thus it can be converted back into a normal function.

If one uses the “dynamic attribute grammar” technique to achieve functions, one can extend the formalism further to permit “procedures” that may write fields as well as read fields. First proposed in our earlier work [Boylend 1996b], procedures allow one to abstract over attribute rules and not just over functional computation. This work used dynamic scheduling. Unlike functions however, a procedure may require several “visits” and thus the prototype implementation described in Section 6 that uses static scheduling does not permit procedures. Parigot et al. describe an implementation technique for implementing multiple visit functions that could be used here.

#### 4. Infinite Fiber Construction

This section describes a construction that expresses the semantics of remote attribute grammars in classical terms. It is called a “fiber construction” because each object which implicitly carries a numbers of separate values (for the fields) is seen as a “rope,” which can be separated into the individual *fibers*.<sup>4</sup> The construction yields an improper attribute grammar with an infinite number of attributes, but nevertheless not only sheds light on the semantics of remote attribute grammars, but also leads to scheduling algorithms based on “fiber approximation.” Since this construction is so central to this article, we now devote several pages to an informal motivation before coming to the precise definition.

A field can be (partially) defined at multiple points, but whenever the field is read, only the final, collected, result is produced. Thus, we need some point at which all the partial definitions of a field are collected together; the obvious and elegant solution is to use the point of the object definition: the rules for the production, which declares the object. Thus as already demonstrated with the field operation dependency graph, all partial definitions are sent back to the point where the object is declared where they are collected and then sent to wherever the object is used so they can be available.

In a classical attribute grammar, all value transmission must be done through attribute rules. Thus to simulate remote attribution in a classical attribute grammar, the field values are transmitted through special attributes that parallel the attributes carrying the object. For example, suppose  $a$  is an attribute that transmits an object, then for every field  $f$ , we add an attribute named  $a\hat{f}$  to transmit partial definitions of  $f$  fields back to the object and add an attribute named  $a\$f$  to transmit the final value of the field  $f$  to wherever the field is

<sup>4</sup>This intuition originates with Rodney Farrow [Farrow 1990].

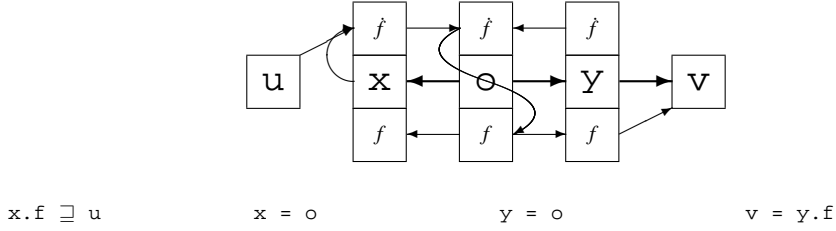


FIG. 8. Simple example of fibering

used. The first new attribute  $a\dot{f}$  transmits values in the opposite direction to  $a$  (if  $a$  is synthesized, then  $a\dot{f}$  is inherited and vice versa) because the partial definitions flow from the uses of the object back to the definition, whereas the attribute  $a f$  transmits values in the same direction as  $a$ . Thus if we have  $n$  fields, then every attribute  $a$  is accompanied by  $n$  attributes going in the opposite direction and  $n$  going in the same direction; these  $2n$  new attributes are called *fibers*. Similarly every local attribute  $l$  induces  $2n$  local attribute fibers.

Let us consider an illustrative example using local attributes for simplicity. Figure 8 gives a graphical view of this same example.

$$\begin{array}{ll}
 x.f \equiv u & x\dot{f} = u \\
 x = o & x = o \\
 & o\dot{f} = x\dot{f} \\
 & o f = o\dot{f} \\
 \\ 
 y = o & y = o \\
 & y\dot{f} = o f \\
 v = y.f & v = y\dot{f}
 \end{array}$$

On the left, we have some remote attribute grammar rules; on the right, we have the fiber reduction of these rules. This example is simplified in several ways (in particular, we should have definitions of  $x\dot{f}$  and  $y\dot{f}$  as well to reflect Fig. 8), but shows that that  $\dot{f}$  fibers are assigned in the opposite way as their base attribute, whereas the  $f$  fibers are assigned in the same way. We call the former fibers *reverse fibers* and the latter fibers *normal fibers*.

The example also shows how we add a rule to define  $o f$  in terms of  $o\dot{f}$ ; this is the collection point, and corresponds to the edge  $o \xrightarrow{f^{-1}} o$  in the field operation dependency graph. Finally it shows how  $v$  is made to depend (indirectly, of course) on  $u$ .

This basic picture is complicated by several additional aspects:

- (1) If an attribute occurrence is used in two places, we end up with multiple definitions of the reverse fibers.
- (2) A single attribute may carry more than one object, perhaps through the use of a primitive conditional or primitive collection (such as a bag).
- (3) An object reference may be stored in a field. Thus we need fibers for the fibers.

These complications require us to define the construction more carefully. The intuition

behind the full construction is described for each case in the following pages.

Item 1 points out that the example should have included the following rule as well:

$$o\$f = y\$f$$

since there might be a partial definition of the  $f$  field through the reference in  $y$  as well as through the one in  $x$ . Unfortunately this means there are multiple definitions of the fiber  $o\$f$ , and no definition of  $y\$f$ . In general, a reverse fiber may be defined any number of times, depending on how many times the base attribute is used. (Normal fibers do not have this problem, since they will be defined exactly as many times as the base attribute is defined, that is, exactly once.) The solution is to form a single definition from the collection of all the various generated fiber definitions. In essence, we use collection assignment. Using collection assignment does not mean remote attribute grammars are defined in terms of themselves, because here the collection assignments are on attributes, not fields, and can always be combined locally. It is a simple matter to locate all the definitions and put them together. Indeed, the main purpose for the fiber constructed attribute grammar is to create dependency graphs, in which case collection assignments form dependencies in the same manner as regular assignments.

Item 2 indicates some of the complications that primitive functions can cause. For example, if we use a primitive conditional such as `if` as follows:

```
x = if(b, o, p)
x.f ⊇ u
v = x.g ,
```

then what definition should we use for  $x\$g$  or  $o\$f$ ? It might seem one would wish to generate something such as:

```
x$g = if(b, o$g, p$g)
o$f ⊇ if(b, x$f, empty())
```

(where `empty()` creates an empty set of definitions). Such a construction, however, would need to know the exact semantics of all primitive operations that can transmit objects. Instead, we define a construction that needs only know which arguments to the primitive functions can transmit objects. In the case of `if`, only the second and third arguments can transmit objects. A fiber then must be able to handle fields of all the different objects that may be transmitted by the base attribute (which may be, for instance, a primitive collection). Thus a fiber will be seen as carrying sets of pairs: the first element of each pair being the object reference to which the field belongs, and the second element being the value or partial value of the field. Thus we generate

```
x$g = fjoin(o$g, p$g)
o$f ⊇ x$f
p$f ⊇ x$f
```

where `fjoin` combines values of fiber attributes. (This can be seen as “union.”)

Similarly the generated rules for creating the definition of  $o\$f$  will include the following:

```
o$f = collect[f](o, o$f)
```

The `collect` function (which is annotated with the actual field being collected) gives the object reference to use as a key and the fiber, which includes the pairs. (The annotation is not needed for creating the dependency graph, but gives information that would be needed to implement the function if we have different initial values and combination functions for different fields.) Only the pairs relevant to the given object are collected. The result is used to create a singleton set with a pair giving the final value of the field  $f$ :

$$\text{collect}[f](o, l) = \{\langle o, \bigcup\{p \mid \langle o, p \rangle \in l\} \rangle\}.$$

Similarly, when we select a field such as in  $v = x.g$ , we replace this line with the following:

$$v = \text{select}[g](x, x\$g)$$

We need to use the value of the object reference (here  $x$ ) as a key to index into the set of pairs (which represent the field selection function) to get the correct value:

$$\text{select}[f](x, l) = \bigcup\{v \mid \langle o, v \rangle \in l, o \in x\}.$$

Unlike our less sophisticated construction above, in this case  $v$  depends directly on  $x$  and not just  $x\$g$ . This additional dependency occurs as  $x \xrightarrow{u} v$  in the field operation dependency graph, along with  $x \xrightarrow{g^{-1}} v$ .

A partial field assignment is the converse. A rule such as  $x.f \sqsubseteq u$  is converted into the following:

$$x\$f \sqsubseteq \text{write}[f](x, u)$$

The rule creates the partial definition fiber keyed by the object  $x$ :

$$\text{write}[f](x, v) = \{\langle o, v \rangle \mid o \in x\}.$$

We need to use  $\sqsubseteq$  in the generated rule because there may be multiple partial definitions of  $x.f$  in the local rule set, but again this is merely a local collection. As before, the fiber depends on the actual object reference, with the corresponding edges in the field operation dependency graph being  $x \xrightarrow{u\ddagger} x, u \xrightarrow{\ddagger} x$ .

Item 3 has a profound effect: fields can themselves carry objects. Suppose we start with rules such as the following:

$$\begin{aligned} q &= x.g \\ b &= q.f \end{aligned}$$

We fetch an object from the  $g$  field of  $x$  and then fetch the  $f$  field from it. We generate the following fiber rules:

$$\begin{aligned} q &= \text{select}[g](x, x\$g) \\ b &= \text{select}[f](q, q\$f) \end{aligned}$$

but the definition for  $q\$f$  has a new form:

$$q\$f = x\$gf$$

In other words, the (possible) value for the  $f$  field of  $q$  is carried by the attribute  $x\$gf$ . There is no need to use a `select` function, and it would not be possible in any case since

$x\$g$  is not an object reference. A deep (and non-obvious) connection to the field operation dependency graph is that the label  $u$  can only be part of  $S$ , not of  $P$ .

The infinitude of the construction is now apparent, since the object in  $x.g$  has all fields, not just  $f$ , and so we need to add

$$q\$g = x\$gg$$

and then the objects in these fields have fields themselves, and thus we need to add rules such as

$$q\$ff = x\$gff$$

$$q\$fg = x\$gfg$$

...

*ad infinitum*. This result holds even in the absence of remote field writes. That is, even if we require all fields to be defined local to the object definition, we still need an infinity of fibers to model the dependencies. But despite the fact that the construction is infinite, it still is useful, so let not the reader be dismayed.

These fibers of fibers also must be transmitted through the base object. Thus, suppose  $o$  is an object declared in the production and  $f$  and  $g$  are fields. We not only add the rules:

$$o\$f = \text{collect}[f](o, o\$f)$$

$$o\$g = \text{collect}[g](o, o\$g)$$

but also the following rules:

$$o\$ff = o\$ff$$

$$o\$fg = o\$fg$$

$$o\$gf = o\$gf$$

⋮

It is not necessary to collect the values, since the fibers such as  $o\$fg$  hold the *final*  $g$  values of objects being collected into the  $f$  field. It is possible for a single object to be placed in the collection for  $f$  more than once, but an object's  $g$  field has the same value at all places.

Fibers such as  $o\$ff$  flow in the reverse direction and originate at partial field definitions. For example the rule  $u.g \sqsupseteq p$  generates not only  $u\$g \sqsupseteq \text{write}[g](u, p)$  but also additional rules to handle the fields of (object(s) referenced by)  $p$ . Somewhere, someone may read the  $g$  field of the objects referenced by  $u$ . Then there may be uses of field  $f$ , and thus the final value of the  $f$  field must be copied back to where the objects in  $u$  are defined. Furthermore, remote field writes may be carried out; the information in those writes comes back here (in parallel with  $u$ ) and is handed off to  $p$ . Thus, the fiber construction generates the following rules for every field  $f$ :

$$u\$gf \sqsupseteq p\$f$$

$$p\$f \sqsupseteq u\$gf$$

The first rule indicates that the final value of  $p$ 's  $f$  field is transported back to where  $u$  gets its objects. The second rule shows that partial definitions of  $p$ 's  $f$  field come back from the point where the objects  $p$  were sent and are added to the partial definitions recorded for  $p$ . These rules demonstrate that  $gf$  is a fiber that travels in the *same* direction as the base

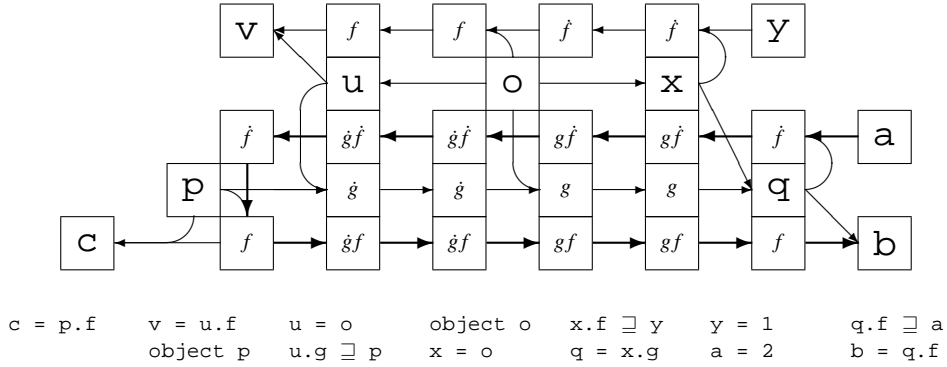


FIG. 9. Fibers for example from Fig. 7 with path from a to b shown.

fiber (a reverse fiber on a reverse fiber is a normal fiber). In the set of rules declaring the object, we thus need to add rules such as the following:

$$o\$\dot{g}\dot{f} = o\$\dot{g}\dot{f}$$

These rules take the partial definitions of the  $f$  field of objects fetched from the final value of the  $g$  field and send them off to the points where  $g$  was (partially) defined so that they can be delivered to the object that needs them. No collection is necessary here since the partial definitions of the  $f$  field will be collected at the point where that object is declared.

Here  $o\$\dot{g}\dot{f}$  is a reverse fiber carrying partial definitions of  $f$  fields of objects fetched from the  $g$  field of the object. Thus whenever we have the rule such as  $q = x.g$ , we need to add the definition of this reverse fiber for  $q$ :

$$x\$\dot{g}\dot{f} \supseteq q\$\dot{f}$$

In other words, the partial definitions of the  $f$  field of the object or objects carried in  $q$  are passed back to the object whose  $g$  field was used to define  $q$ . Then they will be transmitted back to the places where the  $g$  field was defined. The objects that are put in this field are then informed of these partial definitions of  $f$  fields. Figure 9 replays an earlier example (see Fig. 7) to show how it is expressed with fibers. (Only the “relevant” fibers are shown: the ones that actually induce dependencies between the original attributes.) If one compares the two figures, one can see the strong connection between the field operation dependency graph and the dependency graph for the fiber construction. Starting from the base attribute  $a$ , we apply the field operation on each edge to the front of the fiber to get the fiber to be used on the next attribute. Thus (in the outlined path from  $a$  to  $b$ ), we go from  $a$  (with an empty fiber) to  $q\$\dot{f}$ , and then to  $x\$\dot{g}\dot{f}$  (the over-bar is ignored), and then to  $o\$\dot{g}\dot{f}$ , and so on.

The reasoning for the generation of nested fibers can be continued to give meaning to such fibers as  $x\$\dot{f}\dot{g}\dot{h}$ , which is a normal direction fiber carrying partial definitions of  $h$  fields of objects read from  $g$  fields of objects in partial definitions of  $f$  fields. These fibers are used at partial definitions of  $f$  fields, for example  $x.f \supseteq u$  induces the extra rule:

$$u\$\dot{g}\dot{h} = x\$\dot{f}\dot{g}\dot{h}$$

The fibers follow along in parallel as the object is transmitted from its definition to this

point. Back at an object definition for an object  $o$ , we add the following rule defining the fiber:

$$o\$fgh = o\$fgh$$

The reverse fiber  $o\$fgh$  carries partial definitions of  $h$  fields of objects read from the  $g$  fields of objects read from the  $f$  fields. When we find a rule  $w = x.f$ , we add the following fiber rule:

$$x\$fgh \sqsupseteq w\$gh$$

In this way longer fibers are related to shorter fibers.

The precise definition of the infinite fiber construction requires a proper definition of fibers. For a given set of fields  $F$ , the set of *fibers* (denoted  $\Phi$ ) is the set of zero or longer strings of fields and dotted fields:

$$\Phi = \{f, \dot{f} \mid f \in F\}^*$$

the sets of *normal fibers* (denoted  $\Phi^{\rightarrow}$ ) and *reverse fibers* (denoted  $\Phi^{\leftarrow}$ ) are defined as the smallest subsets of  $\Phi$  satisfying the following equalities:

$$\begin{aligned} \Phi^{\rightarrow} &= \{f\phi \mid \phi \in \Phi^{\rightarrow}\} \cup \{\dot{f}\phi \mid \phi \in \Phi^{\leftarrow}\} \cup \{\varepsilon\} \\ \Phi^{\leftarrow} &= \{f\phi \mid \phi \in \Phi^{\leftarrow}\} \cup \{\dot{f}\phi \mid \phi \in \Phi^{\rightarrow}\}. \end{aligned}$$

It follows directly from these definitions that the normal and reverse sets partition the set of fibers:  $\Phi^{\rightarrow} \cup \Phi^{\leftarrow} = \Phi$ ,  $\Phi^{\rightarrow} \cap \Phi^{\leftarrow} = \emptyset$ . The empty fiber  $\varepsilon$  is used to represent the base attribute, that is,  $x\$ \varepsilon$  means the same as  $x$ .

Now follows the precise definition of the infinite fiber construction. The definition of rules makes use of an intermediate form with collections. The *infinite fiber construction* for a remote attribute grammar  $A = (G, S, I, L, B, F, R)$  is an improper classical attribute grammar  $A' = (G, S \times \Phi^{\rightarrow} \cup I \times \Phi^{\leftarrow}, I \times \Phi^{\rightarrow} \cup S \times \Phi^{\leftarrow}, (L \cup B) \times \Phi, R')$  where  $R'$  is defined below. The resulting attribute grammar is improper in that the sets of attributes are all infinite assuming  $F \neq \emptyset$ . The attribute names are pairs; the syntactic sugar  $\$$  operation is used to form pairs as follows:

$$\begin{aligned} l\$ \phi &= (l, \phi) \\ o\$ \phi &= (o, \phi) \\ (X.a)\$ \phi &= X.(a, \phi). \end{aligned}$$

First an intermediate form of the constructed rules is defined:

$$R^{\exists p} = \left\{ \begin{array}{l} (o, \varepsilon) = \text{object}() \quad o \in B^p \\ (o, f) = \text{collect}[f]((o, \varepsilon), (o, \dot{f})) \quad f \in F, o \in B^p \\ (o, f\phi) = (o, \dot{f}\phi) \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, f \in F, o \in B^p \\ (o, \dot{f}\phi) = (o, f\phi) \quad \phi \in \Phi^{\leftarrow}, f \in F, o \in B^p \\ v_0 \$ \varepsilon = g(v_1 \$ \varepsilon, \dots, v_k \$ \varepsilon) \quad (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v_0 \$ \phi = \text{fjoin}(v_i \$ \phi \mid L_{gi} = 0) \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v_i \$ \phi \sqsupseteq v_0 \$ \phi \quad L_{gi} = 0, \phi \in \Phi^{\leftarrow}, (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v \$ \varepsilon = \text{select}[f](w \$ \varepsilon, w \$ f) \quad (v = w.f) \in R^p \\ v \$ \phi = w \$ f\phi \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, (v = w.f) \in R^p \\ w \$ f\phi \sqsupseteq v \$ \phi \quad \phi \in \Phi^{\leftarrow}, (v = w.f) \in R^p \\ w \$ \dot{f} \sqsupseteq \text{write}[f](w \$ \varepsilon, v \$ \varepsilon) \quad (w.f \sqsupseteq v) \in R^p \\ w \$ \dot{f}\phi \sqsupseteq v \$ \phi \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, (w.f \sqsupseteq v) \in R^p \\ v \$ \phi \sqsupseteq w \$ \dot{f}\phi \quad \phi \in \Phi^{\leftarrow}, (w.f \sqsupseteq v) \in R^p \end{array} \right\}.$$

Then  $R'$  is defined for each production from the intermediate form by collecting together multiple definitions as follows

$$R'^p = \left\{ \begin{array}{l} v' = e \quad (v' = e) \in R^{\exists p} \\ v \$ \phi = \text{fjoin}(e \mid (v \$ \phi \sqsupseteq e) \in R^{\exists p}) \quad v \in UO^p, \phi \in \Phi^{\leftarrow} \end{array} \right\}.$$

This set of rules will be well-defined and well-formed as long as the base remote attribute grammar is well-formed. The constructed attribute grammar may be improper (infinite), but we can still form (infinite) dependency graphs and compound dependency graphs.

There is a close connection between the dependency graphs formed from the infinite fiber construction and the field operation dependency graph formed using the original remote attribute grammar. First, formalizing the intuition described earlier, the ‘‘field operations’’ that label the field operation dependency graph can be defined as (partial) functions over fibers:

$$\begin{array}{rcccl} \forall & \phi \in \Phi^{\rightarrow} & \bar{\phi} \in \Phi^{\leftarrow} & f \in F & \\ 0(\phi) & = & \phi & \bar{0}(\bar{\phi}) & = & \bar{\phi} \\ u(\varepsilon) & = & \varepsilon & & & \\ uf(\varepsilon) & = & f & u\dot{f}(\varepsilon) & = & \dot{f} \\ \dot{f}(\phi) & = & \dot{f}\phi & \bar{f}(\bar{\phi}) & = & f\bar{\phi} \\ f^{-1}(f\phi) & = & \phi & \dot{f}^{-1}(\dot{f}\bar{\phi}) & = & \bar{\phi} \\ \dot{f}^{-1}\dot{f}(\dot{f}\phi) & = & f\phi & \bar{f}^{-1}\bar{f}(f\bar{\phi}) & = & \dot{f}\bar{\phi}. \end{array}$$

The  $u$  label can only be applied to the base fiber, and  $\bar{u}$  is undefined for all fibers. A string of labels  $\gamma$  can be applied to a fiber through function composition: for all  $\phi \in \Phi$ ,  $(\gamma_1 \gamma_2)(\phi) = \gamma_2(\gamma_1(\phi))$ ,  $\lambda(\phi) = \phi$ .

Next, it is shown that paths in the compound field operation dependency graph are closely related to paths in the compound dependency graph of the infinite fiber construction:

**LEMMA 4.1.** *Given a remote attribute grammar  $A$  and its infinite fiber construction  $A'$ , and given any  $t$ , form the compound field operation dependency graph  $D_F(t)$ , and the compound dependency graph  $D'(t)$ . Then*

- (1) For every path  $v\$ \phi \xrightarrow{*} v'\$ \phi' \in D'(t)$ , there exists a path  $v \xrightarrow{\gamma} v' \in D_F(t)$  such that  $\phi' = \gamma(\phi)$ .
- (2) Conversely, for every path  $v \xrightarrow{\gamma} v' \in D_F(t)$  with  $\phi' = \gamma(\phi)$ , then there exists a path  $v\$ \phi \xrightarrow{*} v'\$ \phi' \in D'(t)$ .

(Here  $v$  may refer either to an attribute instance or to an object instance.)

PROOF. Stronger results are proved: that the paths are the same length. Because of the definition of composition  $(\gamma_1 \gamma_2)(\phi) = \gamma_2(\gamma_1(\phi))$ , and because the null path corresponds exactly to the null string, we need only prove the result for paths of length one.

- (1) Given  $v\$ \phi \rightarrow v'\$ \phi' \in D'(t)$ , this edge must come from one  $D^p$  for some  $p$ . Each induced dependency edge from the constructed rules  $R^{\neq p}$  is examined in turn, and a labeled edge  $v \xrightarrow{\gamma} v' \in D_F^p$  (see page 22) is found, where  $\gamma$  is a single label.

$$o\$ \varepsilon \rightarrow o\$ f. \quad o \in B^p, \gamma = uf$$

$$o\$ \dot{f} \rightarrow o\$ f. \quad o \in B^p, \gamma = \dot{f}^{-1} f$$

$$o\$ \dot{f} \phi \rightarrow o\$ f \phi. \quad o \in B^p, \varepsilon \neq \phi \in \Phi^{\rightarrow}, \gamma = \dot{f}^{-1} f$$

$$o\$ f \phi \rightarrow o\$ \dot{f} \phi. \quad o \in B^p, \phi \in \Phi^{\leftarrow}, \gamma = \bar{f}^{-1} \dot{f}$$

$$v_i \$ \varepsilon \rightarrow v_0 \$ \varepsilon. \quad v_0 = g(\dots, v_i, \dots) \in R^p, \gamma = L_{gi}$$

$$v_i \$ \phi \rightarrow v_0 \$ \phi. \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, v_0 = g(\dots, v_i, \dots) \in R^p, L_{gi} = 0, \gamma = 0$$

$$v_0 \$ \phi \rightarrow v_i \$ \phi. \quad \phi \in \Phi^{\leftarrow}, v_0 = g(\dots, v_i, \dots) \in R^p, L_{gi} = 0, \gamma = \bar{0}$$

$$w\$ \varepsilon \rightarrow v\$ \varepsilon. \quad v = w.f \in R^p, \gamma = u$$

$$w\$ f \rightarrow v\$ \varepsilon. \quad v = w.f \in R^p, \gamma = f^{-1}$$

$$w\$ f \phi \rightarrow v\$ \phi. \quad \varepsilon \neq \phi \in \Phi^{\rightarrow}, v = w.f \in R^p, \gamma = f^{-1}$$

$$v\$ \phi \rightarrow w\$ f \phi. \quad \phi \in \Phi^{\leftarrow}, v = w.f \in R^p, \gamma = \bar{f}$$

$$w\$ \varepsilon \rightarrow w\$ \dot{f}. \quad (w.f \sqsupseteq v) \in R^p, \gamma = u\dot{f}$$

$$v\$ \varepsilon \rightarrow w\$ \dot{f}. \quad (w.f \sqsupseteq v) \in R^p, \gamma = \dot{f}$$

$$v\$ \phi \rightarrow w\$ \dot{f} \phi. \quad (w.f \sqsupseteq v) \in R^p, \phi \in \Phi^{\rightarrow}, \gamma = \dot{f}$$

$$w\$ \dot{f} \phi \rightarrow v\$ \phi. \quad (w.f \sqsupseteq v) \in R^p, \phi \in \Phi^{\leftarrow}, \gamma = \dot{f}^{-1}$$

- (2) The proof of the second part requires simply that we do the same case analysis in reverse, distinguishing  $\varepsilon$  from other normal fibers to determine which generated rule the dependency comes from.

□

The fiber construction is very regular, even the extra cases for  $\varepsilon$  generate dependencies that *include* the dependencies for other normal fibers. More generally, we have the following rule:

LEMMA 4.2. *Given a remote attribute grammar  $A = (G, S, I, F, L, B, R)$ , let  $A'$  be its infinite fiber construction. If a (compound) dependency graph of  $A'$  has an edge  $v\$ \phi f \rightarrow v'\$ \phi' f$ , then it also has the edge  $v\$ \phi \rightarrow v'\$ \phi'$ . If it has an edge  $v'\$ \phi' \dot{f} \rightarrow v\$ \phi \dot{f}$ , then it also has the edge  $v\$ \phi \rightarrow v'\$ \phi'$ .*

*Conversely, if it has the edge  $v\$ \phi \rightarrow v'\$ \phi'$  where  $\phi, \phi' \in \Phi \setminus \varepsilon$ , then it also has the edges  $v\$ \phi f \rightarrow v'\$ \phi' f$  and  $v'\$ \phi' \dot{f} \rightarrow v\$ \phi \dot{f}$ .*

PROOF. Suppose we have the edge  $v\$ \phi f \rightarrow v'\$ \phi' f$ . This edge must come from one of the rules in a particular production. Since removing  $f$  from the fiber doesn't change its

direction, the same rule should generate rules with the edge  $v\$ \phi \rightarrow v' \$ \phi'$ . The only tricky case is if one or the other of  $\phi$  or  $\phi'$  is  $\varepsilon$ . The  $\varepsilon$  cases, as can be seen by looking at the definition include all the dependencies of the non- $\varepsilon$  cases (as well as others).

Suppose now that we have the edge  $v\$ \phi \hat{f} \rightarrow v' \$ \phi' \hat{f}$ . From the definition of direction, it is clearly seen that  $\phi$  has the opposite direction from  $\phi \hat{f}$  for any  $\phi$ , including  $\phi'$ . From the construction we see that the direction of  $\phi$  determines the direction of the edges in the dependency graph, the same nodes are involved in any case. So removing a  $\hat{f}$  from both fibers means that we will have a generated edge in the reverse direction, hence  $v' \$ \phi' \rightarrow v \$ \phi$ .

The converse part follows directly from the infinite fiber construction and the definition of  $\Phi^{\rightarrow}$  and  $\Phi^{\leftarrow}$ , because of the omission of  $\varepsilon$ .  $\square$

The finiteness of cycles allows us to prove the following lemma that states that we can always assume that a cycle includes an  $\varepsilon$  fiber:

LEMMA 4.3. *Suppose we have a circular remote attribute grammar  $A$ . Let  $A'$  be its infinite fiber construction, and  $t$  be the tree that induces the compound dependency graph  $D'$  of  $A'$  with a cycle  $v_0 \$ \phi_0 \rightarrow v_1 \$ \phi_1 \rightarrow \dots \rightarrow v_n \$ \phi_n = v_0 \$ \phi_0$ , then there exists a cycle in  $D'$  including a node  $v \$ \varepsilon$ .*

PROOF. Assume  $D'$  has a cycle. Without loss of generality, we assume we have chosen the cycle such that  $\phi_0$  be a shortest string that occurs in any cycle in  $D'$ . If  $\phi_0 = \varepsilon$ , we are done. Otherwise, we will determine a contradiction. Suppose  $\phi_0 = f \phi^*$  or  $\hat{f} \phi^*$ ; let  $\phi'_0$  be this single element,  $f$  or  $\hat{f}$ . Then since the dependencies connecting fibers only involve adding or removing a field or dotted field from the front, or replacing a field with its dotted counterpart, or vice versa, and since  $\phi_0$  is a shortest string in the cycle, all strings  $\phi_i$  will be in the form  $\phi'_i \phi^*$ . Now if  $\phi^* \neq \varepsilon$ , we can apply Lemma 4.2 to get a cycle with a shorter fiber, which contradicts our assumption. Furthermore, if all the shortest fibers have the same form, then we can still apply the Lemma and find a cycle starting  $v_0 \$ \varepsilon \rightarrow \dots$

Otherwise we may assume that we have a cycle that includes both nodes of the form  $v \$ f$  and of the form  $v \$ \hat{f}$  and no node of the form  $v \$ \varepsilon$ . We now prove that this assumption leads to a contradiction. Without loss of generality, let  $\phi_0 = f$  and  $\phi_j = \hat{f}$ , and for every  $0 < i < j$  we must have  $|\phi_i| \geq 2$ . Now for any such  $i$ , since no rule can change the ‘‘dotted-ness’’ of a field in a fiber unless it is at the top, and then it stays the same length, we must have that  $\phi_i = \phi f$  for some  $\phi$ . But then it is impossible to see how  $\phi_i = \phi_{j-1}$  can be connected to  $\phi_j$ . Thus no such  $i$  can exist;  $j$  must be 1. But again there is a problem, because there is no way a fiber  $\hat{f}$  can depend directly on a fiber  $f$ , as can be seen by the rules for creating the infinite fiber construction. Thus we have a contradiction, and thus there must exist a cycle with an  $\varepsilon$  fiber.  $\square$

In order to connect the existence of balanced cycles in the compound field operation dependency graph with cycles in the compound dependency graph of the infinite fiber construction, the following lemma about strings of labels is proved:

LEMMA 4.4. *A label string  $\gamma$  maps the empty fiber to itself ( $\gamma(\varepsilon) = \varepsilon$ ) if and only if it can be derived from  $S$  ( $S \xRightarrow{*} \gamma$ ).*

PROOF. A stronger result is proved:

- (1)  $\gamma(\varepsilon) = \varepsilon$  iff  $S \xRightarrow{*} \gamma$ ;
- (2)  $\forall \phi \in \Phi^{\rightarrow} \gamma(\phi) = \phi$  iff  $P \xRightarrow{*} \gamma$ ;

(3)  $\forall \phi \in \Phi^- \gamma(\phi) = \phi$  iff  $\bar{P} \xrightarrow{*} \gamma$ .

$\Rightarrow_{1,2,3}$ . We prove by induction over the length of  $\gamma$ . If  $\gamma$  is empty ( $\gamma = \lambda$ ), the results are true, since  $S \Rightarrow P \Rightarrow \lambda, \bar{P} \Rightarrow \lambda$ . Next assume  $\gamma$  is one element long.

- (1) If  $\gamma(\varepsilon) = \varepsilon$ , then  $\gamma \in \{0, u\}$  which proves our result since  $S \Rightarrow P \Rightarrow 0, S \Rightarrow u$ .
- (2) If  $\gamma(\phi) = \phi$  for every normal fiber ( $\phi \in \Phi^+$ ), then the only possibility is  $\gamma = 0$ , for which we have  $P \Rightarrow 0$ .
- (3) If  $\gamma(\phi) = \phi$  for every reverse fiber ( $\phi \in \Phi^-$ ), then the only possibility is  $\gamma = \bar{0}$ , for which we have  $\bar{P} \Rightarrow \bar{0}$ .

Now suppose  $\gamma$  is composed of two non-empty parts that have the antecedent behavior:  $\gamma = \gamma_1 \gamma_2$  with (case 1)  $\gamma_1(\varepsilon) = \gamma_2(\varepsilon) = \varepsilon$  or (cases 2 and 3)  $\gamma_1(\phi) = \gamma_2(\phi) = \phi$ . Then the result follows by induction since  $S \Rightarrow SS, P \Rightarrow PP, \bar{P} \Rightarrow \bar{P}\bar{P}$ .

Next suppose that  $\gamma$  has no occurrence of a  $u$ ,  $uf$ , or  $uf$  label in it. We consider the cases:

- (1) Because there are no labels that have  $u$  in their name, the functions all apply to an infinite number of fibers in a parallel way, and thus  $\gamma(\varepsilon) = \varepsilon$  implies  $\gamma(\phi) = \phi$  for all normal fibers  $\phi$ . This antecedent implies the antecedent for case 2, and thus using  $S \Rightarrow P$ , we are done, once the next case is proved.
- (2) Consider what the first label of  $\gamma$  is. It must apply to all normal fibers ( $\phi \in \Phi^+$ ). The possibility of 0 would mean that  $\gamma$  is composed of two with the same property which has already been handled. The only remaining possibility is  $\dot{f}$  for some  $f \in F$ . Consider the various fibers resulting from applying the first label of  $\gamma$  to  $\varepsilon$ , the first two labels, the first three labels, and so on until we reach  $\gamma$  itself. None of the results (save the last) can be  $\varepsilon$ , since otherwise the string would be composed. Furthermore, the only way the  $\dot{f}$  can be removed is either to have an edge labeled  $\dot{f}^{-1}f$  in the middle or  $\dot{f}^{-1}$  at the end. The latter case is incompatible with  $\phi \in \Phi^+$ , and so the middle case must exist. In order for it to apply, the result just before must be the fiber  $\dot{f}$  and the fiber just after  $f$ . A similar argument shows that the only way to remove the  $f$  is to end with  $f^{-1}$ . Thus  $\gamma$  has the following shape  $\gamma = \dot{f} \alpha \dot{f}^{-1} f \beta f^{-1}$ . By induction, we achieve  $\bar{P} \xrightarrow{*} \alpha, P \xrightarrow{*} \beta$ , and thus  $P \xrightarrow{*} \gamma$ .
- (3) An analogous argument shows that  $\gamma = \bar{f} \alpha \bar{f}^{-1} \bar{f} \beta \bar{f}^{-1}$  with induction to prove the result.

Next suppose that  $\gamma$  has a  $u$ ,  $uf$ , or  $uf$  label within it. In that case, the antecedents to cases 2 or 3 cannot be satisfied because the composition would not be able to return an infinite range. And  $u$  is not possible in any case, because it would mean we would be able to decompose  $\gamma$  into two parts.

If  $\gamma$  starts with  $uf$ , then that  $f$  can only be removed by a label  $f^{-1}$  at the end (not earlier, or else there would be composition possibility), and thus  $\gamma$  has the form  $\gamma = uf \alpha f^{-1}$ . Since  $\gamma$  is not composed of two, there can be no  $\varepsilon$  in the partial results (applying the first  $n$  labels to  $\varepsilon$ ), and thus we have  $\alpha(f\phi) = f\phi$  for all  $\phi \in \Phi^+$  and furthermore  $\alpha(\varepsilon) = \varepsilon$ , which allows us to use induction to achieve  $P \xrightarrow{*} \alpha$  and thus  $S \xrightarrow{*} \gamma$ .

The case for  $\gamma$  starting with  $uf$  is similar to the case for  $P$ .

$\Leftarrow_{1,2,3}$ . As before, we prove by induction on  $\gamma$ , this time over the derivation of  $\gamma$ . If  $\gamma$  is empty, then the desired results follow immediately since  $\lambda(\phi) = \phi$  for all fibers  $\phi \in \Phi$ . Otherwise, if it consists of a single label, we have the following cases:

$S \Rightarrow P \Rightarrow 0$ . The result is immediate since  $0(\phi) = \phi$  for all normal fibers  $\phi \in \Phi^{\rightarrow}$  including  $\phi = \varepsilon$ .

$S \Rightarrow u$ . The result is immediate:  $u(\varepsilon) = \varepsilon$ .

$\bar{P} \Rightarrow \bar{0}$ . The result is immediate since  $\bar{0}(\phi) = \phi$  for all reverse fibers  $\phi \in \Phi^{\leftarrow}$ .

If  $\gamma$  is derived by a composition,  $S \Rightarrow SS \xrightarrow{*} \gamma$ ,  $P \Rightarrow PP \xrightarrow{*} \gamma$  or  $\gamma \Rightarrow \bar{P}\bar{P} \xrightarrow{*} \gamma$ , then we can divide  $\gamma$  up into two pieces and use induction to reach the desired result.

Otherwise, we consider the remaining cases:

$S \Rightarrow P \xrightarrow{*} \gamma$ . By induction we get result 2, which implies what we need.

$S \Rightarrow ufPf^{-1} \xrightarrow{*} \gamma$ . Thus  $\gamma$  must have the form  $uf\alpha f^{-1}$ , with  $P \xrightarrow{*} \alpha$ . By induction  $\forall \phi \in \Phi^{\rightarrow} \alpha(\phi) = \phi$  and in particular  $\alpha(f) = f$  and thus  $\gamma(\varepsilon) = f^{-1}(\alpha((uf)(\varepsilon))) = \varepsilon$ .

$S \Rightarrow uf\bar{P}\dot{f}^{-1}fPf^{-1} \xrightarrow{*} \gamma$ . Here,  $\gamma$  must have the form  $uf\alpha\dot{f}^{-1}f\beta f^{-1}$ , with  $\bar{P} \xrightarrow{*} \alpha$  and  $P \xrightarrow{*} \beta$ . By induction we get  $\alpha(\dot{f}) = \dot{f}$  and  $\beta(f) = f$  and thus we get  $\gamma(\varepsilon) = f^{-1}(\beta((\dot{f}^{-1}f)(\alpha((uf)(\varepsilon)))))) = \varepsilon$ .

$P \Rightarrow \dot{f}\bar{P}\dot{f}^{-1}fPf^{-1} \xrightarrow{*} \gamma$ . Here  $\gamma$  must have the form  $\dot{f}\alpha\dot{f}^{-1}f\beta f^{-1}$ , with  $\bar{P} \xrightarrow{*} \alpha$  and  $P \xrightarrow{*} \beta$ . By induction we get  $\alpha(\dot{f}\phi) = \dot{f}\phi$  for all normal fibers ( $\phi \in \Phi^{\rightarrow}$ ) and  $\beta(f\phi) = f\phi$  and thus we get  $\gamma(\phi) = f^{-1}(\beta((\dot{f}^{-1}f)(\alpha(\dot{f}(\phi)))))) = \phi$ .

$\bar{P} \Rightarrow \bar{f}\bar{P}\bar{f}^{-1}\dot{f}P\dot{f}^{-1} \xrightarrow{*} \gamma$ . Completely analogous to the previous case.

□

The previous results demonstrate that the infinite fiber construction captures the scheduling constraints of the remote attribute grammar:

**THEOREM 4.5.** *A remote attribute grammar  $A = (G, S, I, L, B, F, R)$  is circular if and only if its infinite fiber construction  $A' = (G, S', I', L', R')$  is circular, that is if for some tree  $t$  generated by the grammar  $G$ , the compound dependency graph of  $A'$  for that tree has a (finite) cycle.*

**PROOF.** If  $A$  is circular, then by Theorem 3.6, there must be a cycle  $v \xrightarrow{S} v$  in the compound field operation dependency graph. By Lemma 4.4, the sequence of labels in this path  $\gamma$  must map the base fiber to itself ( $\gamma(\varepsilon) = \varepsilon$ ). Then by Lemma 4.1, there must exist a (finite) cycle in  $D'$  ( $v\$ \varepsilon \xrightarrow{*} v\$ \varepsilon$ ).

On the other hand, if we have a cycle in  $D'$ , then by Lemma 4.3, we may assume it involves an empty fiber  $v\$ \varepsilon \xrightarrow{*} v\$ \varepsilon$ , and thus we can apply Lemma 4.1 to get a cycle  $v \xrightarrow{\gamma} v$  in  $D_F$  where  $\gamma(\varepsilon) = \varepsilon$ . Now by Lemma 4.4, we have  $S \xrightarrow{*} \gamma$ , and thus it is a *balanced* cycle and thus by Theorem 3.6,  $A$  is circular. □

The infinite fiber construction, while infinite, is nonetheless indirectly useful algorithmically because it can be projected onto a finite partition of the fibers. The resulting (proper) classical attribute grammar can be analyzed using standard classical attribute grammar schedulers. This idea for a finite partition is called “fiber approximation” [Farrow 1990] and is formalized in the following definition:

Given a remote attribute grammar  $A = (G, S, I, L, B, F, R)$  and its infinite fiber construction  $A' = (G, S \times \Phi^{\rightarrow} \cup I \times \Phi^{\leftarrow}, I \times \Phi^{\rightarrow} \cup S \times \Phi^{\leftarrow}, (L \cup B) \times \Phi, R')$ , and a finite partition  $\bar{\Phi} = \{\Phi_1, \dots, \Phi_n\}$  of the set of fibers  $\Phi$ , the *fiber approximation of  $A$  for  $\bar{\Phi}$*  (written  $A/\bar{\Phi}$ ) is the (proper) classical attribute grammar  $A'' = A/\bar{\Phi} = (G, S \times \bar{\Phi}^{\rightarrow} \cup I \times \bar{\Phi}^{\leftarrow}, I \times \bar{\Phi}^{\rightarrow} \cup S \times$

$\bar{\Phi}^{\leftarrow}, (L \cup B) \times \bar{\Phi}, R''$ ) where  $\bar{\Phi}^{\rightarrow} = \{\Phi_i \in \bar{\Phi} \mid \Phi_i \cap \Phi^{\rightarrow} \neq \emptyset\}$  and  $\bar{\Phi}^{\leftarrow} = \{\Phi_i \in \bar{\Phi} \mid \Phi_i \cap \Phi^{\leftarrow} \neq \emptyset\}$ . For every  $v\$ \Phi_i \in DO^p(A'')$  we have the following rule in  $R''$ :

$$v\$ \Phi_i = \text{use}(v\$ \Phi_j \mid v\$ \phi = g(\dots v\$ \phi' \dots) \in R', \phi \in \Phi_i, \phi' \in \Phi_j)$$

The fiber approximation is only used to create dependency graphs and thus uses an uninteresting primitive operation `use`. In other words, any dependency  $v\$ \phi \leftarrow v\$ \phi'$  is mapped into a dependency between each side's approximation.

The definition is only well-formed if  $\bar{\Phi}^{\rightarrow} \cap \bar{\Phi}^{\leftarrow} = \emptyset$ , because otherwise an attribute may be identified as both synthesized and inherited. This article only uses partitions that obey this restriction; in fact, they will be “clean for fibering” as described presently.

**THEOREM 4.6.** *Let  $A = (G, S, I, L, B, F, R)$  be a remote attribute grammar and  $A'$  its infinite fiber construction. Let  $\bar{\Phi} = \{\Phi_1, \dots, \Phi_n\}$  be a partition of  $\Phi$  for which  $\bar{\Phi}^{\rightarrow} \cap \bar{\Phi}^{\leftarrow} = \emptyset$ . Without loss of generality, we assume  $\varepsilon \in \Phi_1(x)$ . Let  $A/\bar{\Phi}$  be the fiber approximation of the infinite fiber construction  $A'$  of  $A$  for this partition. Then if  $A/\bar{\Phi}$  is not circular, then neither is  $A$ . Furthermore, if no cycles in dependency graphs compounded from  $A/\bar{\Phi}$  involve nodes with fibers from  $\Phi_1$ , then  $A$  is not circular.*

**PROOF.** The first part is clear since a cycle in a graph always shows up as a cycle in the quotient graph. The fact that we can ignore cycles that do not include  $\varepsilon$  fibers is a direct result of Lemma 4.3.  $\square$

Of course, in a realistic system, one does not first construct an infinite attribute grammar and then compute a finite quotient, instead one creates the quotient directly. As it happens, under certain conditions of the partition, one can create an approximation that has the same semantics as well as conservatively approximating the dependencies.

We say that a partition  $\bar{\Phi}$  is *clean for fibering* if it satisfies the following three conditions for  $\Phi_i \in \bar{\Phi}, x \in \{f, \hat{f} \mid f \in F\}$ :

$$\begin{aligned} \Phi_1 &= \{\varepsilon\} \\ \Phi_i &\subseteq \Phi^{\rightarrow} \vee \Phi_i \subseteq \Phi^{\leftarrow} \\ \{x\phi \mid \phi \in \Phi_i\} &\subseteq \Phi_j \in \bar{\Phi}. \end{aligned}$$

The first condition ensures that we don't mix up the “real” attribute values with the fiber attributes. The second condition states that every partition consists either of all normal, or all reverse fibers. The last condition ensures that “lengthening” all the fibers in a partition gives us a set that can still be represented by a single element of the partition. When the last condition is satisfied, we write  $x \cdot \Phi_i$  to mean this  $\Phi_j$ .

If the partition is clean for fibering, we define the *clean fiber approximation* that avoids the uses of the infinite construction as an intermediate form. The construction is similar to the infinite fiber construction. The main difference is that we ensure that all attributes defining non-base-fiber attributes are defined using primitive `fjoin`. First we form the

rules:

$$R^{\supseteq p} = \left\{ \begin{array}{l} (o, \Phi_1) = \text{object}() \quad o \in B^p \\ (o, f \cdot \Phi_1) = \text{fjoin}(\text{collect}[f]((o, \Phi_1), (o, \dot{f} \cdot \Phi_1))) \quad f \in F, o \in B^p \\ (o, f \cdot \Phi) = \text{fjoin}((o, \dot{f} \cdot \Phi)) \quad \Phi \in \bar{\Phi}^{\rightarrow}, f \in F, o \in B^p \\ (o, \dot{f} \cdot \Phi) = \text{fjoin}((o, f \cdot \Phi)) \quad \Phi \in \bar{\Phi}^{\leftarrow}, f \in F, o \in B^p \\ v_0 \$ \Phi_1 = g(v_1 \$ \Phi_1, \dots, v_k \$ \Phi_1) \quad (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v_0 \$ \Phi = \text{fjoin}(v_i \$ \Phi \mid L_{gi} = 0) \quad \Phi \in \bar{\Phi}^{\rightarrow}, (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v_i \$ \Phi \sqsupseteq \text{fjoin}(v_0 \$ \Phi) \quad L_{gi} = 0, \Phi \in \bar{\Phi}^{\leftarrow}, (v_0 = g(v_1, \dots, v_k)) \in R^p \\ v \$ \Phi_1 = \text{select}[f](w \$ \Phi_1, w \$ f \cdot \Phi_1) \quad (v = w.f) \in R^p \\ v \$ \Phi = \text{fjoin}(w \$ f \cdot \Phi) \quad \Phi \in \bar{\Phi}^{\rightarrow}, (v = w.f) \in R^p \\ w \$ f \cdot \Phi \sqsupseteq \text{fjoin}(v \$ \Phi) \quad \Phi \in \bar{\Phi}^{\leftarrow}, (v = w.f) \in R^p \\ w \$ \dot{f} \cdot \Phi_1 \sqsupseteq \text{fjoin}(\text{write}[f](w \$ \Phi_1, v \$ \Phi_1)) \quad (w.f \sqsupseteq v) \in R^p \\ w \$ \dot{f} \cdot \Phi \sqsupseteq \text{fjoin}(v \$ \Phi) \quad \Phi \in \bar{\Phi}^{\rightarrow}, (w.f \sqsupseteq v) \in R^p \\ v \$ \Phi \sqsupseteq \text{fjoin}(w \$ \dot{f} \cdot \Phi) \quad \Phi \in \bar{\Phi}^{\leftarrow}, (w.f \sqsupseteq v) \in R^p \end{array} \right\}$$

where

$$\begin{aligned} \bar{\Phi}^{\rightarrow} &= \{\Phi \in \bar{\Phi} \mid \Phi \cap \Phi^{\rightarrow} \neq \emptyset, \Phi \neq \Phi_1\} \\ \bar{\Phi}^{\leftarrow} &= \{\Phi \in \bar{\Phi} \mid \Phi \cap \Phi^{\leftarrow} \neq \emptyset\}. \end{aligned}$$

Then, as with the finite fiber construction, we collect together multiple definitions of reverse fibers:

$$R'^p = \left\{ \begin{array}{l} v' = e \quad (v' = e) \in R^{\supseteq p} \\ v \$ \Phi = \text{fjoin}(e \mid (v \$ \Phi \sqsupseteq \text{fjoin}(\dots, e, \dots))) \in R^{\supseteq p} \quad v \in UO^p, \Phi \in \bar{\Phi}^{\leftarrow} \end{array} \right\}.$$

This set of rules will be well-defined and well-formed<sup>5</sup> as long as the base remote attribute grammar is well-formed and the partition is clean for fibering. The first condition ensures that the base fiber can be used to carry the actual value of an attribute. The second condition ensures that we can determine whether  $X.a \$ \Phi_i$  is inherited or synthesized. The third condition makes the use of the “.” operator possible in the construction.

LEMMA 4.7. *If the partition is clean for fibering, the clean fiber approximation generates the same dependency graphs as the fiber approximation.*

PROOF. Immediate by comparison of the two constructions.  $\square$

This result together with the result about scheduling makes implementation possible once we have a partition. In our earlier work [Boyland 1998], we assume the partition is specified by the programmer in terms of annotations. Even a partition is not enough for practical implementation because the approximation constructions add new attributes even when the fibers make no sense: one cannot fetch a field from an integer, for instance. These extra dependencies, when obscured by approximation, not only lead to a slower analysis (more attributes) but also to more frequent finding of cycles. Thus we want a way to determine and remove the “irrelevant fibers” that can never produce useful information. The following section thus develops a “relevant fiber” analysis, which helpfully produces a partition as well.

<sup>5</sup>The astute reader will notice that this construction generates some rules of the nonstandard form, e.g.,  $v_0 = g(\dots, g'(v_i, v_{i+1}), \dots)$  where  $g = \text{fjoin}$  and  $g' = \text{write}[f]$ , but these can easily be interpreted as uses of new primitive functions  $g''$  with the nesting flattened out.

### 5. Analysis

Given that detecting circularity for remote attribute grammars is undecidable, this section examines techniques for approximation. First it is shown that many fibers are not *relevant* to the final schedule. Then a constraint system for computing (a superset of) the relevant fibers is described. This system is then converted into a practical “two stage” form, in which the second stage can be expressed as a finite-state automaton recognizing relevant fibers. Finally this automaton is used to generate partitions that are clean for fibering.

**5.1. RELEVANT FIBERS.** The approximation theorem requires one to partition the whole set of fibers including those which may never be involved in dependencies between base fibers. Only the latter fibers are relevant to circularity. The approximation, however, may be forced to combine irrelevant fibers in ways that obscure the fact they are never involved in such dependencies. Thus before one forms an approximation, it makes sense to pare down the set of fibers to those that may occur in a cycle involving a base fiber. This will not solve the undecidability problem of the previous section; it only prevents our approximation from being needlessly imprecise.

Let  $A = (G, S, I, L, B, F, R)$  be a remote attribute grammar and  $A'$  its infinite fiber construction. Let  $v \in O^p$  be an attribute occurrence of a production  $p$  in  $G$ , then  $\Phi_P(v)$  (the *provided fibers of v*) is the set of all fibers  $\phi \in \Phi$  such that there exists a tree  $t$  over  $G$  including an instance of  $p$  and the compound dependency graph  $D'$  of  $A'$  for this tree  $t$  includes a path for some  $w$

$$w\$ \mathcal{E} \rightarrow \dots \rightarrow v\$ \phi$$

where in a slight abuse of notation,  $v$  is also used to refer to any instance of  $v$  in  $D'$ . Similarly the *required fibers of v*,  $\Phi_R(v)$  is the set of all fibers  $\phi \in \Phi$  such that there exists  $t$  and a  $w$  such that  $D'$  includes a path

$$v\$ \phi \rightarrow \dots \rightarrow w\$ \mathcal{E}.$$

Third, the *relevant fibers for v*,  $\Phi(v)$ , are the set of all fibers  $\phi \in \Phi$  such that there exists a tree  $t$  over  $G$  and two instances  $w$  and  $w'$  and the compound dependency graph  $D'$  of  $A'$  for this tree  $t$  includes a path

$$w\$ \mathcal{E} \rightarrow \dots \rightarrow v\$ \phi \rightarrow \dots \rightarrow w' \$ \mathcal{E}.$$

Finally, these sets are extended to apply to attributes as well as attribute occurrences:

$$\Phi_P(a) = \bigcup_{X.a \in O^p, p \in P} \Phi_P(X.a)$$

$$\Phi_R(a) = \bigcup_{X.a \in O^p, p \in P} \Phi_R(X.a)$$

$$\Phi(a) = \bigcup_{X.a \in O^p, p \in P} \Phi(X.a)$$

This lemma immediately follows from these definitions:

**LEMMA 5.1.** *Given  $A = (G, S, I, L, B, F, R)$  a remote attribute grammar and  $v \in O^p$  be an attribute occurrence of a production  $p$  in  $G$ , then the set of relevant fibers is never more than the intersection of the provided and required fibers:*

$$\Phi(v) \subseteq \Phi_P(v) \cap \Phi_R(v)$$

*The same relation holds for attributes as well.*

There is not always equality because a fiber is only relevant if it is provided and required for the *same* tree  $t$  and same instance of  $v$ .

The relevant fiber set can be used to generate a “smaller” (but potentially still infinite) fiber construction:

Given a remote attribute grammar  $A = (G, S, I, F, L, B, R)$  and a set  $\Phi^\sharp(x)$  for every attribute occurrence or attribute  $x$ , the *relevant fiber construction* of  $A$  is a potentially improper classical attribute grammar  $A'' = A^{\Phi^\sharp} = (G, S', I', L', R'')$  where

$$\begin{aligned} S'(X) &= \{(a, \phi) \mid a \in S(X), \phi \in \Phi^\sharp(a) \cap \Phi^\rightarrow\} \cup \{(a, \phi) \mid a \in I(X), \phi \in \Phi^\sharp(a) \cap \Phi^\leftarrow\} \quad , \\ I'(X) &= \{(a, \phi) \mid a \in I(X), \phi \in \Phi^\sharp(a) \cap \Phi^\rightarrow\} \cup \{(a, \phi) \mid a \in S(X), \phi \in \Phi^\sharp(a) \cap \Phi^\leftarrow\} \quad , \\ L' &= \{(l, \phi) \mid l \in L \cup B, \phi \in \Phi^\sharp(l)\} \quad , \end{aligned}$$

and  $R''$  is constructed from  $R'$  in the infinite fiber construction by only including rules for the defined occurrences in the new attribute grammar and only including uses of the used occurrences in the new attribute grammar. (Other rules and uses are removed.)

The relevant fiber construction captures the important dependency paths in the remote attribute grammar, as expressed in the following lemma:

LEMMA 5.2. *Let  $A$  be a remote attribute grammar and  $A'$  and  $A'' = A^{\Phi^\sharp}$  be its infinite fiber construction and the relevant fiber construction using  $\Phi^\sharp(x) \supseteq \Phi(x)$  respectively. Let  $t$  be some tree described by the grammar and  $D'$  and  $D''$  be the compound dependency graphs in  $A'$  and  $A''$  respectively. Finally let  $v_0\$ \phi_0 \rightarrow \dots v_n\$ \phi_n$  be some non-trivial path ( $n > 0$ ) where  $\phi_0 = \phi_n = \varepsilon$ . This path appears in  $D'$  if and only if it appears in  $D''$ .*

PROOF. Suppose there is a non-trivial path  $v_0\$ \phi_0 \rightarrow \dots v_n\$ \phi_n$  in  $D'$  where  $\phi_0 = \phi_n = \varepsilon$ . By definition of  $\Phi(x)$ , the relevant fibers of  $x$ , we know  $\phi_i \in \Phi(v_i)$  for all  $i$  ( $0 \leq i \leq n$ ). Furthermore, for any  $v_i$  of the form  $X.a$ , we know  $\phi_i \in \Phi(a)$ . For every edge  $v_i\$ \phi_i \rightarrow v_{i+1}\$ \phi_{i+1}$  in  $D'$ , this edge must have come from a rule in  $A'$  as defined in the infinite fiber construction. The relevant fiber construction preserves dependencies between relevant attribute occurrences, and thus this edge will also appear in  $D''$ . Conversely, we only remove dependencies when constructing  $A''$ , and thus every edge in  $D''$  is also in  $D'$ .  $\square$

From this lemma follows immediately the following lemma, which states that the relevant fiber construction captures circularity.

LEMMA 5.3. *Given a remote attribute grammar  $A$  and a set  $\Phi^\sharp(x)$  for every attribute and attribute occurrence  $x$ , which includes all relevant fibers ( $\Phi^\sharp(x) \supseteq \Phi(x)$ ). Let  $A^{\Phi^\sharp}$  be the relevant fiber construction of  $A$ . Then  $A$  is circular if and only if there exists a tree  $t$  for which the compound dependency graph  $D'$  formed using  $A^{\Phi^\sharp}$  has a cycle including an attribute base fiber instance  $v\$ \varepsilon$ .*

PROOF. First we handle the direction ( $\Leftarrow$ ). If there is such a cycle in  $D'$ , then by Lemma 5.2 it also occurs in the infinite fiber construction, and thus  $A$  is circular by definition.

Suppose now that  $A$  is circular. Then by Lemma 4.3, there must exist a tree  $t$  for which the infinite fiber construction of  $A$  induces a compound dependency graph with a cycle  $v_0\$ \phi_0 \rightarrow v_1\$ \phi_1 \rightarrow \dots \rightarrow v_n\$ \phi_n = v_0\$ \phi_0$  including a node  $v\$ \varepsilon$ . Without loss of generality,

$\phi_0 = \phi_n = \varepsilon$ . Then applying Lemma 5.2 determines that this cycle must also be in the relevant fiber construction.  $\square$

We combine this result with fiber approximation by first defining the combined construction and then proving that it safely approximates true circularity.

Given a remote attribute grammar  $A = (G, S, I, F, L, B, R)$ , finite partition  $\bar{\Phi} = \{\Phi_1, \dots, \Phi_n\}$  (where  $\varepsilon \in \Phi_1$ ) and a set  $\Phi^\sharp(x)$  for every attribute occurrence or attribute  $x$ , the *relevant fiber approximation* of  $A$  using  $\Phi^\sharp$  and  $\bar{\Phi}$  is a (proper) classical attribute grammar  $A^{\Phi^\sharp}/\bar{\Phi} = (G, S', I', L', R')$  where

$$\begin{aligned} S'(X) &= \{(a, \Phi) \mid a \in S(X), \Phi \cap \Phi^\sharp(a) \cap \Phi^\leftarrow \neq \emptyset\} \cup \{(a, \Phi) \mid a \in I(X), \Phi \cap \Phi^\sharp(a) \cap \Phi^\leftarrow \neq \emptyset\} \ , \\ I'(X) &= \{(a, \Phi) \mid a \in I(X), \Phi \cap \Phi^\sharp(a) \cap \Phi^\leftarrow \neq \emptyset\} \cup \{(a, \Phi) \mid a \in S(X), \Phi \cap \Phi^\sharp(a) \cap \Phi^\leftarrow \neq \emptyset\} \ , \\ L' &= \{(l, \Phi) \mid l \in L \cup B, \Phi \cap \Phi^\sharp(l) \neq \emptyset\} \ , \end{aligned}$$

and  $R'$  is constructed from  $R$  in the infinite fiber construction by constructing the following rule for each  $v\$ \Phi \in DO^p(A'')$ :

$$\begin{aligned} v\$ \Phi &= \text{use}(v' \$ \Phi' \in UO^p(A'') \mid v\$ \phi = g(\dots v' \$ \phi' \dots) \in R', \\ &\quad \phi \in \Phi \cap \Phi^\sharp(v), \phi' \in \Phi' \cap \Phi^\sharp(v')) \ . \end{aligned}$$

**THEOREM 5.4.** *Let  $A = (G, S, I, L, B, F, R)$  be a remote attribute grammar, and  $\bar{\Phi} = \{\Phi_1, \dots, \Phi_n\}$  be a finite partition of the set of fibers  $\Phi$  where  $\varepsilon \in \Phi_1$ . For any attribute, local attribute or object  $x$ , let the set  $\Phi^\sharp(x)$  include the relevant fibers of  $x$ :  $\Phi^\sharp(x) \supseteq \Phi(x)$ . Let  $A'' = A^{\Phi^\sharp}/\bar{\Phi}$  be the relevant fiber approximation of  $A$  using  $\Phi^\sharp$ . Then if  $A''$  is not circular, then neither is  $A$ . Furthermore, if no cycles in dependency graphs compounded from  $A''$  involve nodes with fibers from  $\Phi_1$ , then  $A$  is not circular.*

**PROOF.** Suppose  $A''$  is not circular. Then the relevant fiber construction  $A^{\Phi^\sharp}$  is not circular because dependency graphs in  $A''$  are quotient graphs of  $A^{\Phi^\sharp}$ . Therefore, by Lemma 5.3,  $A$  is not circular either. The second part also follows from Lemma 5.3.  $\square$

This theorem is used in the following conservative algorithm for determining noncircularity of a remote attribute grammar:

- (1) Find an approximation  $\Phi^\sharp(x)$  of the relevant set of fibers for every attribute, local attribute or object  $x$ .
- (2) Find a finite partition  $\bar{\Phi}$  of the fibers.
- (3) Construct  $A^{\Phi^\sharp}/\bar{\Phi}$ , the relevant fiber approximation.
- (4) Perform a standard evaluation-class test (such as SNC [Courcelle and Franchi-Zanettacci 1982] or OAG [Kastens 1980]), except that we keep an additional bit for each summary edge—whether that summary edge may include a “base fiber” attribute.
- (5) Report a cycle only if a circularity is found that may include a base fiber attribute. (Section 6 describes how cycles involving only non-base fibers can be cut.)

**5.2. FIBER ANALYSIS.** The rest of this section describes static analysis methods for computing safe approximations to the relevant sets and determining a partition. The relevant fibers are computed by first computing an (over-approximation) to the provided and required fibers and then intersecting the sets. One may see the provided fibers as “flowing” from an occurrence of an epsilon fiber and the required fibers as “flowing” to such an

$$\begin{array}{ll}
a \in A(X), X \in N. & X.a \in UO^p. \\
F(a) \supseteq \{\varepsilon\} & F(X.a) \supseteq F(a) \\
\bar{F}(a) \supseteq \{\varepsilon\} & \bar{F}(a) \supseteq \bar{F}(X.a) \\
v \in O^p. & X.a \in DO^p. \\
F(v) \supseteq \{\varepsilon\} & F(a) \supseteq F(X.a) \\
\bar{F}(v) \supseteq \{\varepsilon\} & \bar{F}(X.a) \supseteq \bar{F}(a) \\
o \in B^p. & v_0 = g(\dots v_i \dots) \text{ where } L_{g_i} = 0. \\
F(o) \supseteq \{f \mid f \in F\} & F(v_0) \supseteq F(v_i) \\
F(o) \supseteq \{f\phi \mid f\phi \in \bar{F}(o)\} & \bar{F}(v_i) \supseteq \bar{F}(v_0) \\
F(o) \supseteq \{\dot{f}\phi \mid f\phi \in \bar{F}(o)\} & v = w.f. \\
w.f \supseteq v. & F(v) \supseteq \{\phi \mid f\phi \in F(w)\} \\
\bar{F}(w) \supseteq \{\dot{f}\phi \mid \phi \in F(v)\} & \bar{F}(w) \supseteq \{f\phi \mid \phi \in \bar{F}(v)\} \\
\bar{F}(v) \supseteq \{\phi \mid \dot{f}\phi \in F(w)\} &
\end{array}$$

FIG. 10. Fiber analysis

occurrence. In other words, the provided fibers follow the data-flow through the attribute grammar, whereas required fibers follow it in the reverse direction. This situation is complicated by the fact that reverse fibers (those in  $\Phi^{\leftarrow}$ ) have the opposite data-flow direction as the normal fibers (those in  $\Phi^{\rightarrow}$ ) which parallel the base attribute. As a result, the analysis computes sets of “provided normal” fibers at the same time as the “required reverse” fibers.

For instance, if there is the equation  $v = w$ , then if  $w$  has a provided fiber  $\phi$  and if  $\phi \in \Phi^{\rightarrow}$  is a normal fiber, then that means there is some path starting with an epsilon fibered attribute occurrence that ends with  $w\$ \phi$ . But since the infinite fiber construction includes the rule  $v\$ \phi = w\$ \phi$ , we can extend the path by one step and discover that  $\phi$  will be a provided fiber of  $v$ . Conversely, suppose  $\tilde{\phi} \in \Phi^{\leftarrow}$  is a required reverse fiber of  $w$ , that is there is some path starting at  $w\$ \tilde{\phi}$  that ends in an epsilon fibered attribute occurrence. Then since  $w\$ \tilde{\phi}$  is defined partly in terms of  $v\$ \tilde{\phi}$ , the latter attribute occurrence could also start a path towards an epsilon fibered attribute occurrence. Thus  $\tilde{\phi}$  is seen to be a required fiber of  $v$ .

Our analysis therefore computes the set of provided normal fibers of each attribute occurrence together with the set of required reverse fibers (the combined set is called  $F(v)$ ), and similarly the set of provided reverse fibers with the set of required normal fibers ( $\bar{F}(v)$ ).

Figure 10 specifies constraints induced by a remote attribute grammar. Any solution to these constraints (in particular, the least solution) is a conservative approximation to the relevant sets; some irrelevant fibers may be included, but no relevant fibers are omitted, as stated in the following lemma:

LEMMA 5.5. *Given  $A$ , a remote attribute grammar, and sets  $F(v)$  and  $\bar{F}(v)$  for every attribute occurrence  $v$  that satisfy the constraints in Figure 10, then*

$$\Phi(v) \subseteq F(v) \cap \bar{F}(v).$$

PROOF. By Lemma 5.1, it suffices to show

$$\begin{aligned}\Phi_P(v) &\subseteq (F(v) \cap \Phi^{\rightarrow}) \cup (\bar{F}(v) \cap \Phi^{\leftarrow}) \\ \Phi_R(v) &\subseteq (F(v) \cap \Phi^{\leftarrow}) \cup (\bar{F}(v) \cap \Phi^{\rightarrow}).\end{aligned}$$

Given a fiber  $\phi \in \Phi_P(v)$ , we show by induction on the length of the path  $w\$ \varepsilon \rightarrow \dots \rightarrow v\$ \phi$  that  $\phi$  is in the set  $(F(v) \cap \Phi^{\rightarrow}) \cup (\bar{F}(v) \cap \Phi^{\leftarrow})$ .

If the length of the path is zero, that is,  $v = w$ ,  $\phi = \varepsilon$ , then the result follows at once since Figure 10 includes the constraint  $F(v) \supseteq \{\varepsilon\}$ .

If the length of the path is greater than zero, there is a path of the form  $w\$ \varepsilon \rightarrow \dots \rightarrow w' \$ \phi' \rightarrow v \$ \phi$ . If  $\phi = \varepsilon$ , we are done immediately for the same reason as for the base case. Otherwise, by induction, the following must be true:

$$\phi' \in (F(w') \cap \Phi^{\rightarrow}) \cup (\bar{F}(w') \cap \Phi^{\leftarrow}).$$

Now because attributes of a node are identified even as the node plays different roles in its own and its parent production, a single attribute instance may instantiate two attribute occurrences (potentially of different productions). Let  $w'_1, w'_2$  be the two attribute occurrences that are instantiated in  $w'$ , where  $w'_1 \$ \phi'$  is a defined occurrence of  $A'$ , the infinite fiber construction of  $A$ , and  $w'_2 \$ \phi'$  is a used occurrence. If  $w'_1 \neq w'_2$ , then it must be the case that  $w'_1 = X_1.a$ ,  $w'_2 = X_2.a$  for some  $X_1, X_2$  and  $a$ . Now, by induction, the lemma must be true for either  $w'_1$  or  $w'_2$ . If it is true for  $w'_2$ , we can continue. If it is true for  $w'_1$ , then by the rules of Figure 10, either  $F(w'_2) \supseteq F(a) \supseteq F(w'_1)$  if  $\phi \in \Phi^{\rightarrow}$ , or  $\bar{F}(w'_2) \supseteq F(a) \supseteq \bar{F}(w'_1)$  if  $\phi \in \Phi^{\leftarrow}$ , and thus the result is true for  $w'_2$  as well.

Now consider the edge  $w' \$ \phi' \rightarrow v \$ \phi$ . This must be the edge from some instantiation of a rule  $r \in R^{\exists p}$  (see page 33) for some  $p \in P$ . The left-hand side of  $r$  is  $v \$ \phi$  and the right-hand side contains a use of  $w' \$ \phi$ . We now look at each case for  $r$  where  $\phi \neq \varepsilon$ :

$(o, f) = \text{collect}[f]((o\$ \varepsilon), (o, \dot{f}))$  In this case  $v = w = o$ ,  $\phi = f$ , and  $\phi' = \dot{f}$  or  $\phi' = \varepsilon$ . In any case we are done because of the rule  $F(o) \supseteq \{f \mid f \in F\}$ .

$(o, f\phi'') = (o, \dot{f}\phi'')$ ,  $\phi'' \in \Phi^{\rightarrow}$  Thus  $v = w = o$ ,  $\phi = f\phi''$ , and  $\phi' = \dot{f}\phi''$ . Since  $\phi'' \in \Phi^{\rightarrow}$ , then  $\phi' \in \Phi^{\leftarrow}$  and thus by induction  $\phi' = \dot{f}\phi'' \in \bar{F}(o)$ . Then by Figure 10,  $F(o) \supseteq \{f\phi''\}$  and we are done.

$(o, \dot{f}\phi'') = (o, f\phi'')$ ,  $\phi'' \in \Phi^{\leftarrow}$  Here  $v = w = o$ ,  $\phi = \dot{f}\phi'' \in \Phi^{\rightarrow}$ ,  $\phi' = f\phi'' \in \Phi^{\leftarrow}$ . By induction,  $\phi' = f\phi'' \in \bar{F}(o)$ , and then by Figure 10,  $\phi = \dot{f}\phi'' \in F(o)$  and we are done.

$v\$ \phi = \text{fjoin}(\dots w\$ \phi \dots)$ ,  $\phi \in \Phi^{\rightarrow}$  In this case,  $v_0 = v = g(\dots v_i = w' \dots)$  in  $A$  where  $L_{gi} = 0$ . By induction,  $\phi \in F(w')$  and so by Figure 10,  $\phi \in F(v)$  and we are done.

$v\$ \phi \supseteq w' \$ \phi$ ,  $\phi \in \Phi^{\leftarrow}$  In this case  $v_0 = w' = g(\dots v_i = v \dots)$  in  $A$  where  $L_{gi} = 0$ . By induction,  $\phi \in \bar{F}(w')$  (since  $\phi$  is a reverse fiber) and thus by Figure 10,  $\phi \in \bar{F}(v)$  and we are done.

$v\$ \phi = w' \$ f\phi$ ,  $\phi \in \Phi^{\rightarrow}$  The rule is the result of a rule  $v = w.f$  in  $A$ . Here  $\phi' = f\phi \in \Phi^{\rightarrow}$  and thus by induction  $\phi' \in F(w')$ . Then by Figure 10,  $\phi \in F(v)$  and we are done.

$v\$ \phi = v\$ f\phi' \supseteq w' \$ \phi'$ ,  $\phi' \in \Phi^{\leftarrow}$  This is the result of a rule  $w' = v.f$  in  $A$ . Thus by induction,  $\phi' \in \bar{F}(w')$ . Then by Figure 10,  $\phi \in \bar{F}(v)$  and we are done.

$v\$ \phi = v\$ \dot{f}\phi' \supseteq w' \$ \phi'$ ,  $\phi' \in \Phi^{\rightarrow}$  This rule comes from a rule  $v.f \supseteq w'$  in  $A$ . By induction,  $\phi' \in F(w')$ , and thus by Figure 10,  $\phi = \dot{f}\phi' \in \bar{F}(v)$  and we are done.

**Stage 1:**

$o \in B^p, p \in P.$

$$B^\sharp(o) \subseteq \{o\}$$

$X.a \in UO^p.$

$$\begin{aligned} B^\sharp(X.a) &\supseteq B^\sharp(a) \\ \bar{B}^\sharp(a) &\supseteq \bar{B}^\sharp(X.a) \end{aligned}$$

$X.a \in DO^p.$

$$\begin{aligned} B^\sharp(a) &\supseteq B^\sharp(X.a) \\ \bar{B}^\sharp(X.a) &\supseteq \bar{B}^\sharp(a) \end{aligned}$$

$v_0 = g(v_1, \dots, v_n), L_{gi} = 0.$

$$\begin{aligned} B^\sharp(v_0) &\supseteq B^\sharp(v_i) \\ \bar{B}^\sharp(v_i) &\supseteq \bar{B}^\sharp(v_0) \end{aligned}$$

$v = w.f.$

$$\begin{aligned} B^\sharp(v) &\supseteq B^\sharp(v') \mid (f, v') \in \bar{B}^\sharp(o), o \in B^\sharp(w) \\ \bar{B}^\sharp(w) &\supseteq \{(f, v)\} \end{aligned}$$

$w.f \sqsupseteq v.$

$$\begin{aligned} \bar{B}^\sharp(w) &\supseteq \{(f, v)\} \\ \bar{B}^\sharp(v) &\supseteq \bar{B}^\sharp(v') \mid (f, v') \in \bar{B}^\sharp(o), o \in B^\sharp(w) \end{aligned}$$

**Stage 2:**

$o \in \{B^p \mid p \in P\}.$

$$F(o) = \{\varepsilon\} \cup \{f \mid f \in F\} \cup \bigcup_{u \in \bar{B}^\sharp(o)} F(u)$$

where

$$\begin{aligned} F((f, v)) &= \{f\phi \mid \phi \in F(v)\} \\ F((f, \bar{v})) &= \{f\bar{\phi} \mid \bar{\phi} \in \bar{F}(v)\} \end{aligned}$$

$x \in \{O^p - B^p \mid p \in P\} \cup \{a \in A(X) \mid X \in N\}.$

$$F(x) = \{\varepsilon\} \cup \bigcup_{o \in \bar{B}^\sharp(x)} F(o)$$

$x \in \{O^p \mid p \in P\} \cup \{a \in A(X) \mid X \in N\}.$

$$\bar{F}(x) = \{\varepsilon\} \cup \bigcup_{u \in \bar{B}^\sharp(x)} \bar{F}(u)$$

where

$$\begin{aligned} \bar{F}((f, v)) &= \{f\bar{\phi} \mid \bar{\phi} \in \bar{F}(v)\} \\ \bar{F}((f, \bar{v})) &= \{f\phi \mid \phi \in F(v)\} \end{aligned}$$

FIG. 11. Two-stage fiber analysis

$v\$\phi \sqsupseteq w'\$f\phi, \phi \in \Phi^{\leftarrow}$  In this case,  $w'.f \sqsupseteq v$  in  $A$ , and thus since  $\phi' = f\phi \in \Phi^{\rightarrow}$ , then by induction  $\phi' \in F(w')$ . Then by Figure 10,  $\phi \in \bar{F}(v)$  and we are done.

The proof for  $\Phi_R(v)$  is analogous.  $\square$

5.3. TWO-STAGE FIBER ANALYSIS. The constraints in Figure 10 are not easily solvable in finite time. The sets  $F(v)$  and  $\bar{F}(v)$  can both be infinite, containing fibers of unbounded length. It is possible to represent these sets finitely, since (as shown later), they are *regular* (that is, is equal to the yield of a finite state automaton). However, the way in which the constraints are expressed (where fibers get longer or shorter) do not immediately allow a regular representation. Thus a “two-stage” analysis is introduced for computing relevant fibers, as seen in Figure 11. The first stage computes sets of object declarations  $B^\sharp(x)$  for each attribute and attribute occurrence  $x$ . This is the set of objects the instances of which are carried by instances of  $x$ , the objects that could appear in the instances’ values. All instances created from the same object declaration are collapsed into one. The second set  $\bar{B}^\sharp(x)$  computes the sets of rules that either read or write fields from objects carried by  $x$ . Since there are a finite set of object declarations, and a finite set of rules for the remote attribute grammar, this analysis can be computed in finite time, in fact in time  $O(n^3)$  where  $n$  is the size of the remote attribute grammar. The least solution to the equations is used.

The second stage uses the first stage to avoid the need to consider fibers getting shorter. The  $B^\sharp(x)$  sets have essentially short-circuited the process. The  $F(v)$  and  $\bar{F}(v)$  sets can be computed using only copy rules and lengthening. The second stage rules can be expressed as a finite state automaton with states for each object declaration and each rule that reads or writes a field. The fact that the two stage analysis has the same least solution as the original set of constraints in Figure 10 is proved in Lemma 5.7. Then Theorem 5.8 proves the regularity result.

The proof uses an “instrumented” version of Figure 10’s analysis that computes sets of *instrumented fibers* in which objects and local attributes surround the fields:

$$\begin{aligned}\Phi_I^{\rightarrow} &= \{^o f \phi_I \mid \phi_I \in \Phi_I^{\rightarrow}\} \cup \{^o \dot{f} \phi_I \mid \phi_I \in \Phi_I^{\leftarrow}\} \cup \{\lambda\} \\ \Phi_I^{\leftarrow} &= \{^v f \phi_I \mid \phi_I \in \Phi_I^{\leftarrow}\} \cup \{^v \dot{f} \phi_I \mid \phi_I \in \Phi_I^{\rightarrow}\}.\end{aligned}$$

Here  $o$  and  $v$  refer to any object or local attribute in the attribute grammar. Let  $\phi_I|_{\Phi}$  be the uninstrumented fiber corresponding to  $\phi_I$ , that is, with the objects and locals removed. Let  $\lambda$  be an object that does not appear in the attribute grammar. Figure 12 gives the constraints for the instrumented analysis. The least solution to these constraints is used.

The instrumented analysis carries within it the solution to the  $\bar{B}^\sharp$  and  $B^\sharp$  analysis of Figure 11 as shown by the following lemma:

LEMMA 5.6. *As long as  $F \neq \emptyset$ , the solutions to the first stage of Figure 11 are given by the following equations:*

$$\begin{aligned}B^\sharp(x) = B_I^\sharp(x) &= \{o \mid ^o f \phi_I \in F_I(x)\} \\ \bar{B}^\sharp(x) = \bar{B}_I^\sharp(x) &= \{(\dot{f}, v) \mid ^v \dot{f} \phi_I \in \bar{F}_I(x)\} \cup \{(f, v) \mid ^v f \phi_I \in \bar{F}_I(x)\}.\end{aligned}$$

PROOF. We prove this result by showing that the constraints are isomorphic for each syntactic entity. We prove by induction on the uses of  $\bar{B}^\sharp$  and  $B^\sharp$  in creating constraints.

First, we can (at this point) ignore the initialization that places  $\lambda$  in every  $F_I$  and  $\bar{F}_I$  set since this entry has no direct effect on the equations to be proved. In what follows, we use the property that every  $F_I$  and  $\bar{F}_I$  set is non-empty.

Next, note that the equation for  $B_I^\sharp$  above depends on  $F_I$  monotonically and on nothing else. Similarly, that for  $\bar{B}_I^\sharp$  depends on  $\bar{F}_I$  monotonically and on nothing else. Therefore the constraints for  $X.a \in UO^p$ ,  $X.a \in DO^p$ , and  $v_0 = g(\dots v_i \dots)$  are isomorphic.

$$\begin{array}{ll}
a \in A(X), X \in N. & X.a \in UO^p. \\
F_I(a) \supseteq \{\lambda\} & F_I(X.a) \supseteq F_I(a) \\
\bar{F}_I(a) \supseteq \{\lambda\} & \bar{F}_I(a) \supseteq \bar{F}_I(X.a) \\
v \in O^p. & X.a \in DO^p. \\
F_I(v) \supseteq \{\lambda\} & F_I(a) \supseteq F_I(X.a) \\
\bar{F}_I(v) \supseteq \{\lambda\} & \bar{F}_I(X.a) \supseteq \bar{F}_I(a) \\
o \in B^p. & v_0 = g(\dots v_i \dots) \text{ where } L_{g_i} = 0. \\
F_I(o) \supseteq \{^o f^\lambda \mid f \in F\} & F_I(v_0) \supseteq F_I(v_i) \\
F_I(o) \supseteq \{^o f \phi_I \mid ^v \dot{f} \phi_I \in \bar{F}_I(o)\} & \bar{F}_I(v_i) \supseteq \bar{F}_I(v_0) \\
F_I(o) \supseteq \{^o \dot{f} \phi_I \mid ^v f \phi_I \in \bar{F}_I(o)\} & \\
w.f \sqsupseteq v. & \\
\bar{F}_I(w) \supseteq \{^v \dot{f} \phi_I \mid \phi_I \in F_I(v)\} & F_I(v) \supseteq \{\phi_I \mid ^o f \phi_I \in F_I(w)\} \\
\bar{F}_I(v) \supseteq \{\phi_I \mid ^o \dot{f} \phi_I \in F_I(w)\} & \bar{F}_I(w) \supseteq \{^v f \phi_I \mid \phi_I \in \bar{F}_I(v)\}
\end{array}$$

FIG. 12. Instrumented fiber analysis

We need thus only look at the “interesting” cases: objects, field writes and field reads:

$o \in B^p$ . Figure 11 says  $B^\sharp(o) \supseteq \{o\}$ , which is exactly the effect of the constraints in Figure 12: the first line adds at least one (since  $F \neq \emptyset$ ) instrumented fiber of the form  $^o f^\lambda$ , which ensures  $\{o \neq \lambda \mid ^o f \phi_I \in F_I(o)\} \supseteq \{o\}$ . The other two constraints add instrumented fibers of a form not used in the equations for this Lemma. Thus they have exactly the same effect (in constraining  $\bar{B}^\sharp$  and  $B^\sharp$  sets).

$w.f \sqsupseteq v$ . The constraint from Figure 12:  $\bar{F}_I(w) \supseteq \{^v \dot{f} \phi_I \mid \phi_I \in F_I(v)\}$  has exactly the same constraint on  $\bar{B}_I^\sharp(w)$  as  $\bar{B}^\sharp(w) \supseteq \{(f, v)\}$  from Figure 11 since  $F_I(v)$  is never empty.

We can rewrite the next constraint using the instrumentation since each instrumented fiber can only be created in a single place:

$$\begin{aligned}
\bar{F}_I(v) &\supseteq \{\phi_I \mid ^o \dot{f} \phi_I \in F_I(w)\} \\
\bar{F}_I(v) &\supseteq \{\phi_I \mid ^o \dot{f} \phi_I \in F_I(w), ^v f \phi_I \in \bar{F}_I(o)\} \\
\bar{F}_I(v) &\supseteq \{\phi_I \mid ^o \dot{f} \phi_I \in F_I(w), ^v f \phi_I \in \bar{F}_I(o), \phi_I \in \bar{F}_I(v')\} \\
\bar{F}_I(v) &\supseteq \{\phi_I \mid (o \in B^\sharp(w) \wedge (f, v') \in \bar{B}^\sharp(o)), (f, v') \in \bar{B}^\sharp(o), \phi_I \in F_I(v')\}.
\end{aligned}$$

The logical “and” (written  $\wedge$ ) in the last rewrite of the constraint is used because the constraint  $o \in B^\sharp(w)$  is too weak by itself to be equivalent to  $^o \dot{f} \phi_I \in F_I(w)$ , but as one can see, the strengthening is redundant. The final form, when used to compute  $\bar{B}_I^\sharp(v)$  is equivalent to the constraint for  $\bar{B}^\sharp(v)$  from Figure 11.

$v = w.f$ . As with the previous case, we choose the simpler constraint first:  $\bar{F}_I(w) \supseteq \{^v f \phi_I \mid \phi_I \in \bar{F}_I(v)\}$  is equivalent (for the purposes of computing  $\bar{B}_I^\sharp(w)$ ) to the constraint  $\bar{B}^\sharp(w) \supseteq \{(f, v)\}$  since  $\bar{F}_I(v)$  is never empty.

The other constraint can (as before) be rewritten:

$$\begin{aligned}
F_I(v) &\supseteq \{\phi_I \mid ^o f \phi_I \in F_I(w)\} \\
F_I(v) &\supseteq \{\phi_I \mid ^o f \phi_I \in F_I(w), ^v \dot{f} \in \bar{F}_I(o)\}
\end{aligned}$$

$$\begin{aligned}
F_I(v) &\supseteq \{\phi_I \mid {}^o f \phi_I \in F_I(w), {}^{v'} \dot{f} \in \bar{F}_I(o), \phi_I \in F_I(v)\} \\
F_I(v) &\supseteq \{\phi_I \mid (o \in B_I^\sharp(w) \wedge (f, v') \in \bar{B}^\sharp(o)), (f, v') \in \bar{B}_I^\sharp(o), \phi_I \in F_I(v)\}
\end{aligned}$$

This final rewrite of the constraint is equivalent to the one for  $B^\sharp(v)$  from Figure 11.

Since the constraints are equivalent,  $B^\sharp(x) = \bar{B}_I^\sharp(x)$  and  $\bar{B}^\sharp(x) = \bar{B}_I^\sharp(x)$ , which was to be proved.  $\square$

Now we are ready to prove the equivalence of the two analyses:

LEMMA 5.7. *Given a remote attribute grammar  $A$ , the two stage constraints in Figure 11 have the same least solution for  $F(x)$  and  $\bar{F}(x)$  as the constraints in Figure 10.*

PROOF. First, since the constraints of Figure 12 are completely isomorphic to the constraints of Figure 10, it only remains to show the equivalence of the sets in Figure 11 with the uninstrumented results of the analysis of Figure 12. Thus  $F(x)$  and  $\bar{F}(x)$  in this proof are defined by their definitions from Figure 11.

We prove the equivalence by induction over the constraints, showing that the constraints are equivalent in each place, assuming that they are equivalent elsewhere.

First we show how the constraints concerning  $F(a)$  and  $\bar{F}(a)$  are equivalent to those concerning  $F_I(a)$  and  $\bar{F}_I(a)$  (modulo instrumentation). Suppose we have performed the two stage analysis, then for  $X.a \in DO^p$

$$F(a) = \{\varepsilon\} \cup \bigcup_{o \in B^\sharp(a)} F(o) \supseteq \{\varepsilon\} \cup \bigcup_{o \in B^\sharp(X.a)} F(o) = F(X.a)$$

(where the middle inequality comes from the first stage) and thus the results of the two-stage fiber analysis satisfy the constraints of the single-stage analysis. Analogously for  $X.a \in UO^p$ ,

$$F(a) = \{\varepsilon\} \cup \bigcup_{o \in B^\sharp(a)} F(o) \subseteq \{\varepsilon\} \cup \bigcup_{o \in B^\sharp(X.a)} F(o) = F(X.a).$$

The cases for  $\bar{F}(a)$  are analogous. Conversely, suppose  $X.a \in DO^p$ . Then considering the constraint  $F_I(a) \supseteq F_I(X.a)$ , assume the equivalence holds for  $F_I(X.a)$ , that it is equivalent to  $\{\varepsilon\} \cup \bigcup_{o \in B^\sharp(X.a)} F(o)$ . These are the only constraints on  $F_I(a)$  and thus in a (minimal) solution:

$$F_I(a)|_\Phi = \bigcup_{o \in B^\sharp(X.a), X.a \in DO^p} F(o).$$

But  $B^\sharp(a) = \bigcup_{X.a \in DO^p} B^\sharp(X.a)$  and thus we get the required equivalence. The cases for other uses of  $F(a)$  and  $\bar{F}(a)$  are analogous.

The monotonic constraints for  $B^\sharp$  and  $\bar{B}^\sharp$  for  $v_0 = g(v_1, \dots, v_n)$ ,  $L_{gi} = 0$  can similarly be seen as isomorphic to the constraints for  $F_i$  and  $\bar{F}_i$  for the purposes of computing  $F$  and  $\bar{F}$ .

The case for objects  $o \in B^p$  is more interesting. We compute  $F(o)$  as the union of four sets (if we distinguish the union of the  $F((\dot{f}, v))$  from the union of the  $F((f, v))$ ). Correspondingly, we have four constraints for  $F_I(o)$ . The first two constraints ( $F_I(o) \supseteq \{\lambda\}$  and  $F_I(o) \supseteq \{{}^o f^\lambda \mid f \in F\}$ ) are obvious analogues of the first two sets making up the

union. The third constraint is transformed into the third set in the union as follows:

$$\begin{aligned}
F_I(o) &\supseteq \{ {}^o f \phi_I \mid {}^v \dot{f} \phi_I \in \bar{F}_I(o) \} \\
&= \{ {}^o f \phi_I \mid {}^v \dot{f} \phi_I' \in \bar{F}_I(o), {}^v \dot{f} \phi_I \text{ exists}, {}^v \dot{f} \phi_I \in \bar{F}_I(o) \} \\
&= \{ {}^o f \phi_I \mid (\dot{f}, v) \in \bar{B}^\sharp(o), \phi_I \in F_I(v), {}^v \dot{f} \phi_I \in \bar{F}_I(o) \} \\
&= \{ {}^o f \phi_I \mid (\dot{f}, v) \in \bar{B}^\sharp(o), \phi_I \in F_I(v) \}.
\end{aligned}$$

The final equality requires some explanation:  $\subseteq$  is obvious, and  $\supseteq$  follows because the constraint  $(\dot{f}, v) \in \bar{B}^\sharp(o)$  implies that there exists at least *one* instrumented fiber of the form  ${}^v \dot{f} \phi_I'$  in  $\bar{F}_I(o)$  and the constraints for the instrumented analysis transmit all fibers of this form together, and thus in particular  ${}^v \dot{f} \phi_I \in \bar{F}_I(o)$ . The fourth constraint is similarly analogous to the fourth set in the union.

There remains one more case for  $F_I(v)$ , the case where  $v = w \cdot f$ . For this case we have two constraints that we put together into an equality:

$$\begin{aligned}
F_I(v) &= \{ \varepsilon \} \cup \{ \phi_I \mid {}^o f \phi_I \in F_I(w) \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid {}^o f \phi_I' \in F_I(w), {}^o f \phi_I \in F_I(o) \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid o \in B^\sharp(w), {}^v \dot{f} \phi_I \in \bar{F}_I(o) \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid o \in B^\sharp(w), (\dot{f}, v') \in \bar{B}^\sharp(o), {}^v \dot{f} \phi_I \text{ exists} \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid o \in B^\sharp(w), (\dot{f}, v') \in \bar{B}^\sharp(o), \phi_I \in F_I(v') \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid o \in B^\sharp(w), (\dot{f}, v') \in \bar{B}^\sharp(o), o' \in B^\sharp(v'), \phi_I \in F_I(o') \} \\
&= \{ \varepsilon \} \cup \{ \phi_I \mid o' \in B^\sharp(v), \phi_I \in F_I(o') \} \\
&= \{ \varepsilon \} \cup \bigcup_{o' \in B^\sharp(v)} F_I(o').
\end{aligned}$$

In the sixth equality, we use the inductive hypothesis on  $F_I(v')$ . The seventh equality uses the constraint for  $B^\sharp(v)$  (which is effectively an equality constraint since each variable may be defined just once).

If we look at  $\bar{F}(w)$  for this rule, we see that the constraint  $\bar{B}^\sharp(w) \supseteq \{(f, v)\}$  along with the definition of  $\bar{F}((f, v))$  used to form  $\bar{F}(w)$  leads to the inequality  $\bar{F}(w) \supseteq \{f \phi \mid \phi \in \bar{F}(v)\}$ , which is completely analogous to the constraint for  $F_I(w)$  from the instrumented analysis.

The constraints for  $\bar{F}_I(w)$  and  $\bar{F}_I(v)$  for the case  $w \cdot f \supseteq v$  are similarly analogous to the realization of the computations of  $\bar{F}(w)$  and  $\bar{F}(v)$ . Thus we omit these cases.  $\square$

The two-stage analysis may look more complex than the original fiber analysis, but it is much easier to implement. As previously noted, the  $B^\sharp$  and  $\bar{B}^\sharp$  sets can be computed in cubic times. Furthermore, one can construct a finite state automaton that recognizes the  $F$  and  $\bar{F}$  sets as seen in Figure 13.

**THEOREM 5.8.** *The sets  $F(x)$  and  $\bar{F}(x)$  (and their intersection) are regular sets over the alphabet of fields and dotted fields  $\Sigma = \{f, \dot{f} \mid f \in F\}$ .*

**PROOF.** We construct a finite state automaton (Figure 13) with two states  $q_x$  and  $\bar{q}_x$  (representing  $F(x)$  and  $\bar{F}(x)$  respectively) for every  $x \in O^+$  where  $O^+$  is the set of all attribute occurrences and attributes ( $O^+ = \{O^p \mid p \in P\} \cup \{a \in A(X) \mid X \in N\}$ ) and states  $q_u$  and  $\bar{q}_u$  (representing  $F(u)$  and  $\bar{F}(u)$ ) for every  $u \in U$ , where  $U$  is the set of all things that occur in  $\bar{B}^\sharp(x)$  for some  $x$  ( $U = \{u \in \bar{B}^\sharp(x) \mid x \in O^+\}$ ).

$$\begin{aligned}
M &= (Q, \Sigma, \delta, q_0, \Omega) \\
Q &= \{q_u, \bar{q}_u \mid u \in \bar{B}^\sharp(x), x \in O^+\} \cup \Omega \\
\Omega &= \{q_x, \bar{q}_x \mid x \in O^+\} \cup \{q_f \mid f \in F\} \\
\Sigma &= \{f, \hat{f} \mid f \in F\} \\
\delta &= \{q_x \xrightarrow{\varepsilon} q_o \mid x \in O^+, o \in B^\sharp(x)\} \cup \{\bar{q}_x \xrightarrow{\varepsilon} \bar{q}_u \mid x \in O^+, u \in \bar{B}^\sharp(x)\} \cup \\
&\quad \{q_o \xrightarrow{f} q_f \mid o \in B^p, p \in P, f \in F\} \cup \{q_o \xrightarrow{\varepsilon} q_u \mid o \in B^p, p \in P, u \in \bar{B}^\sharp(o)\} \cup \\
&\quad \{\bar{q}_{(f,v)} \xrightarrow{\hat{f}} \bar{q}_v \mid v=w \cdot f \in R^p\} \cup \{\bar{q}_{(f,v)} \xrightarrow{\hat{f}} q_v \mid w \cdot f \sqsupseteq v \in R^p\} \cup \\
&\quad \{q_{(f,v)} \xrightarrow{\hat{f}} \bar{q}_v \mid v=w \cdot f \in R^p\} \cup \{q_{(f,v)} \xrightarrow{\hat{f}} q_v \mid w \cdot f \sqsupseteq v \in R^p\} \\
q_0 &= \begin{cases} q_x & \text{We wish to compute } F(x) \\ \bar{q}_x & \text{We wish to compute } \bar{F}(x) \end{cases}
\end{aligned}$$

FIG. 13. Finite-State Automaton to match  $F$  and  $\bar{F}$  sets.

The automaton is constructed such that with one exception  $q \xrightarrow{\varepsilon} q'$  if and only if  $S \supseteq S'$  is a constraint in the second stage of the fiber analysis where  $q$  is the state corresponding to  $S$  and  $q'$  is the state corresponding to  $S'$ . The exception is that the automaton includes cycles of the form  $q_o \xrightarrow{\varepsilon} q_o$  for each object (since  $o \in B^\sharp(o)$ ), but these cycles have no effect on the strings accepted. Similarly  $q \xrightarrow{y} q'$  if and only if  $S \supseteq \{y\phi \mid \phi \in S'\}$  is a constraint. Furthermore  $q$  is final ( $q \in \Omega$ ) if and only if its corresponding set  $S$  has the constraint  $S \supseteq \{\varepsilon\}$ .

Thus if one starts from any state  $q$  and continues through the automaton stopping at a final state, one will generate a fiber in the corresponding set  $S$ . The converse is also true: any fiber  $\phi \in S$  leads to a final state (possibly after  $\varepsilon$ -moves) if one starts at the corresponding state  $q$ .

Since (nondeterministic) finite-state automata accept only regular languages, it follows that  $F(x)$  and  $\bar{F}(x)$  and their intersection are all regular sets.  $\square$

5.4. USING THE FIBER AUTOMATON. One can use a set-of-subsets construction followed by an automaton intersection algorithm to create an FSA for an approximation  $\Phi^\sharp(x)$  for each attribute or occurrence  $x$ . But then that still leaves the creation of the partition that should be clean for fibering.

Thus instead, we form the deterministic *reverse* automaton for matching fibers. The states of this reverse automaton are sets of states of the original automaton. The “first” state (state 0 in Figure 14, page 52) will be the set of all final states in the original automaton, and will be the one matching the base fiber. Since there are no edges out of each  $q_f$ , the “first” state is the only one which will include them, this state will never be entered again on a reverse matching. Thus if we see the automaton as partitioning the fibers, the first set  $\Phi_1$  will include only  $\{\varepsilon\}$ . Then, we work “backwards” from each deterministic state, finding all states that could reach it by traversing a particular field (or dotted field). New states are added, and we continue the process. The result is similar to that of the standard set-of-subsets construction except that we ensure that each deterministic state matches only normal or only reverse fibers. Furthermore, we avoid reuse of deterministic states (even when the sets of original automaton states are the same) except when a set of states is reachable from itself. The resulting deterministic automaton is far from optimally small,

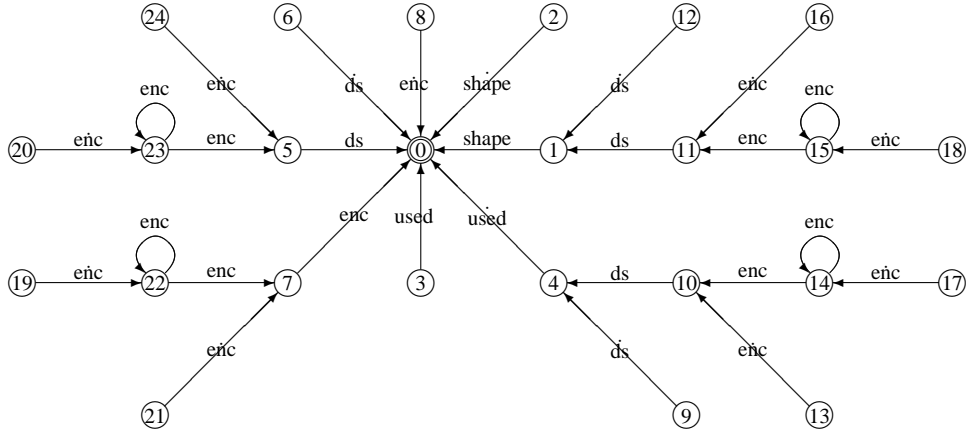


FIG. 14. Reverse FSA for the clean fiber partition.

but finer distinctions avoid spurious circularities in the fibered attribute grammars. In any case, the new automaton is deterministic *in reverse* and thus extending a fiber at the front is guaranteed to go to only one other state, satisfying the third condition for “clean for fibering.”

Now a fiber is in  $\Phi^\sharp(x)$  if and only if both  $q_x$  and  $\bar{q}_x$  are in the state reached when executing the reverse of the fiber in the reverse automaton. Thus the relevant fibers for  $x$  will be defined by a subset of the partition. This permits relevancy and the partition to be represented compactly. For example, Figure 14 gives the reverse automaton for the example in Figure 2. In the fiber approximation, each attribute can have theoretically up to 25 fibers (including the base fiber). However, most have far fewer: for example, the `scope` attribute of `decl` and `decls` has  $\Phi^\sharp = \{0, 6, 9, 12\}$  and the `scope` attribute of `stmt` and `stmts` has  $\Phi^\sharp = \{0, 5, 7, 10, 11, 14, 15, 22, 23\}$ . Only the local `contour` has more:  $\Phi^\sharp = \{0, 5-24\}$ .

One might notice that there are no fibers for the global collection `msgs`, which as explained earlier, is implemented as a (collection) field on a special object passed down from the root. Since there are no uses of this collection, the collection assignments can be assigned as late as the scheduler desires, and thus there is no need to add a scheduling constraint.<sup>6</sup>

Because we avoid sharing in the generation of the deterministic automaton, its shape depends very little on the actual way analysis is performed. The only major contribution is determining when a cycle must be introduced. We experimented with other techniques to produce a smaller deterministic automaton. Folding together two sets of the same states worked fine with the simple example here, but in our larger example (see Section 6.3) it led to spurious circularity. Any attempt to minimize the deterministic automaton must address the fact that it may cause such a problem, which may be difficult for an attribute grammar writer to comprehend, let alone fix. At best, one can provide a way for the writer to “crank up” the size of the automaton subject to practical constraints, such as virtual mem-

<sup>6</sup>In our tool, however, we force the fibers to appear through the introduction of a “fake” use. This permits the scheduler to assume the existence of the fiber when scheduling collection writes.

ory needed to perform the analysis of the fiber approximation attribute grammar. Avoiding this dilemma entirely is impossible given the undecidability of circularity of remote attribute grammars.

In any case, in our limited experience (described in the following section), we have found the large deterministic automata practical. More experience is needed to see if additional heuristics are needed and how they can be communicated to the analysis.

## 6. Implementation and Experiences

It is beyond the scope of this article to describe the implementation in great detail. The reader is referred to an earlier paper [Boyland 2002], which describes in detail how remote attribute grammars are implemented, both for batch and incremental evaluation. Rather, this section will broadly sketch how implementation is carried out; in particular how the theory and analysis of the preceding sections are realized, starting with scheduling and then moving to code generation. It also describes some experiences defining and using the implementation.

6.1. SCHEDULING. The scheduling of a remote attribute grammar is the realization of the analysis described in the previous sections. As such, most of the details have already been explored. Here we merely give a broad-brush description of the scheduling as implemented in our prototype tool.

The attribute grammar is represented in APS, a polymorphically-typed side-effect-free language with built-in support for tree nodes, matching and attributes [Boyland 1996b]. Before scheduling *per se* takes place, the attribute grammar undergoes binding and type inference, and is rejected if type or binding errors are found. The tool determines the attributes for each nonterminal (“phylum” in APS terminology, borrowing from earlier work [Borras et al. 1988; Reps and Teitelbaum 1989; Maddox 1997]) and rules for each production (“constructor”). It also determines the set of fields. Since the tool implements conditional attribute grammars [Boyland 1996a] as well, the tool also determines the number of conditionals (both `if` and `case` statements generate conditionals) to be used to mark dependency edges.

We next apply the first stage of the fiber analysis using a routine that does least fixed-point analysis of  $B^\sharp$  and  $\bar{B}^\sharp$  sets stored on the “fibered declarations” (attributes, locals, objects, formal parameters and return values). Every expression is analyzed by “passing” it a  $\bar{B}^\sharp$  set, which “produces” an  $B^\sharp$  set. We connect the rules for field reads and writes so that we do not need to store  $\bar{B}^\sharp$  sets on field reads:

$$v = w.f.$$

$$\begin{aligned} B^\sharp(v) &\supseteq B^\sharp(v') & (f, v') \in \bar{B}^\sharp(o), o \in B^\sharp(w) \\ \bar{B}^\sharp(v') &\supseteq \bar{B}^\sharp(v) \end{aligned}$$

$$\bar{B}^\sharp(w) \supseteq \{(f, v)\}$$

$$w.f \supseteq v.$$

$$\bar{B}^\sharp(w) \supseteq \{(f, v)\}.$$

In our implementation, the  $\bar{B}^\sharp$  sets are sets of field read expression nodes and field write rule nodes.

Next we form the automata as described in Figure 13 and construct the deterministic backwards finite-state automaton to achieve a partition of all the fibers. We also record which fibers (or rather fiber partitions) are relevant for which attributes. We then form the (classical) dependency graph for the relevant fiber approximation with additional edge markings: all dependencies are marked with their conditionality (under what combination of condition values the dependency exists) and whether the edge is solely between nonbase fibers. We then perform a DNC closure test while preserving the edge markings on indirect dependencies. Conditionality is removed when an edge is copied into the summary graph for a nonterminal. If any cycles are determined that are not solely among nonbase fiber attributes, the remote attribute grammar is rejected.

If cycles remain, we use union-find to find all strongly-connected regions of fibered attributes. For every nonterminal whose fibered attributes take part in the cycle, we create two new attributes, an “up” attribute and a “down” attribute. These attributes (and two new locals at the “top” of the cycle) are used to break the cycle: all elements of the strongly-connected region are removed; all dependencies into the strongly-connected region are moved to the “up” attribute of the left-hand-side nonterminal (or local); and all dependencies out of the strongly-connected region are moved to come from the “down” attribute of the left-hand-side nonterminal (or local).<sup>7</sup> The resulting dependency graphs are resubmitted to the DNC test, which should now complete successfully.

Next, the dependency graphs are submitted to the OAG test to find a schedule ordering. If none is found, the remote attribute grammar is rejected. Such a “type-3 circularity” can be difficult for the user to know what to do with. In our experience, they occur when our tool is presented with an incomplete remote attribute grammar. (It may also be appropriate to perform the full NP-complete search for an order, although this is made less practical because of the large number of attributes after fiberings.) Once the order is found, a conditional total order of the attribute instances in each production is found and this is used as the basis of code generation.

The first stage of the fiber analysis is a minor variant on “dynamic transitive closure” and thus can be solved in cubic time. The (reverse) deterministic FSA is however exponential in the worst-case. Thus the number of attributes scheduled by the classical scheduler can be exponential in the number of fields.

6.2. CODE GENERATION. One advantage of a static schedule is that evaluation takes place in linear time. Since the scheduler has assured that the rules will be executed in a legal order, it is not necessary to test this property at run-time (a worst-time cubic analysis).

Our prototype tool uses a simple visit-sequence evaluation with attributes stored on nodes. The total conditional order is implemented in a number of “visits” for each production. The incremental model used is a fairly standard multiple-edit-site model, which marks the spine from every change site to the root to enable relatively cheap coordination.

The definition of circularity for a remote attribute grammar only concerns the relative ordering of the original rules. The rules generated by the fibers are irrelevant. Thus the fiber attributes become merely “control attributes,” which carry no value and only constrain the scheduling. Once scheduling is done, they can be ignored. Thus at this point, we need only be concerned with implementing the original rules.

The normal attribution rules show up in the relevant fiber approximation dependency

<sup>7</sup>This idea comes from Maddox [1997].

graph at the point where the left-hand-side is scheduled. The same is true for a field read. Field writes, however, do not define any regular attribute, only a field. Thus in the schedule, a field write  $w.f \sqsupseteq v$  is located at the point in the conditional total order where the fibered attribute  $w\$\Phi$  (where  $\Phi \ni f$ ) is scheduled.<sup>8</sup>

Objects are treated as special local attributes initialized at their schedule point to a newly allocated object. In our prototype, objects are indeed locals initialized to the results of a constructor call. Objects are treated the same way as tree nodes: their fields are implemented as attributes (and indeed in our prototype, as declared as APS attributes).

In APS, a field may only be written remotely if it is declared as a “collection” and provides the combination function. The other fields may only be assigned at the place where the object is created. Thus when we come up to the point in a schedule where we need to schedule the  $f$  fiber for a fibered declaration, we find all writes in the current rule that contribute to this fiber, and generate code for them at this point. A noncollection field will have at most one write, and thus a write can be implemented as simply storing the value of the field. A partial write is implemented in the batch system by simply updating the field. In the incremental system, a partial write adds itself to a balanced tree of partial writes for the field of that object. A field read is implemented by simply reading the value of the field. In the incremental system, the read is recorded by including its production instance in a list of “users.”

In the incremental system, writes must be retractable. A noncollection write is retracted by reverting to the default value. A collection write is retracted by removing it from the balanced tree of writes. When a write is added or retracted, the object’s production is marked as an edit site forcing incremental evaluation to visit this production. Upon the visit, if the field has changed value, all the users’ nodes are marked as edit sites, so they can be visited as well. The details are described in an earlier paper [Boylund 2002].

6.3. EXPERIENCES. Our limited experience includes working with the example in Fig. 2 (and variants) and also a larger example: a static semantics for a simple object-oriented language “Cool” (a dialect of Aiken’s Cool [Aiken 1996]) with classes, fields, methods, overriding, local variables, and a few simple expressions. As Java, Cool does not require definition before use, and thus particular care must be used to detect cyclic inheritance so that the type-checker does not get into an infinite loop.

The abstract CFG has thirty productions plus five list nonterminals (each with three productions) for a total of forty-five productions. The remote attribute grammar has sixteen attributes, twelve fields, and two global collections (the class table, and the error messages). The remote attribute grammar is roughly seven times bigger overall than the one in Fig. 2.

When the Cool remote attribute grammar is analyzed, we end up with a partition of 144 fibers, of which no attribute has more than 65 fibers. Two of the production dependency graphs (the ones for class and method declarations) end up with more than 350 nodes. Eight strongly-connected cycles of fiber attributes are found and cut. The resulting schedule generates six visits for the top node of the program:

- (1) Each class is added to the class table.
- (2) Each class gets the parent class field set.

<sup>8</sup>This technique requires, of course, that the fibered attribute exist, thus demonstrating the reason why our analysis implementation generates “fake” uses for each field.

- (3) An artificial pass that ensures that one can follow the parent class field arbitrarily many times.
- (4) Each class has a field set determining whether it is involved with cyclic inheritance. Also the list of “attributes” (fields) of the class is determined.
- (5) An artificial pass ensuring that cyclic inheritance check is done for all classes.
- (6) All the errors are generated.

Two of the passes are artificial and are caused only by the way we break cycles between fibers. To avoid the inefficiency, one could either remove them in a post-pass or investigate how to cut cycles without forcing a pass. Most nonterminals have far fewer visits: The “features” of the class are traversed only twice, and the bodies of methods visited only once. The resulting generated Java batch implementation consists of about 12,000 “words” in about 2500 lines of code, compared to the handwritten C++ semantic analyzer which is also approximately 2500 lines long (albeit including comments).

In summary, most of the implementation details are outside of the scope of this article, but a broad overview gives one a sense of the practicality of the theory described here. In particular we have shown that these techniques can be used to analyze a medium-size “semantic analyzer” for a small, but nontoy language. Indirectly, this also shows some practical implementation results for conditional attribute grammars. We find that while greatly increasing the number of attributes, “fibers” still permit realistic scheduling with the great benefit that the attribute grammar writer can work at a higher level.

### 7. Related Work

LIGA [Kastens 1991], part of the Eli compiler construction system [Gray et al. 1992], 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 [Kastens and Waite 1994]. 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. Using control attributes to schedule imperative actions only defines a safe evaluation order, it does not define when a value (such as a symbol table) is “complete.” This property makes such an attribute grammar difficult to incrementalize.

The Ag attribute grammar implementation tool in the Cocktail toolkit permits remote access to attributes (remote reads) [Grosch 1990]. If static scheduling is used, the attribute grammar writer must ensure (using artificial dependencies, if necessary) that such attributes are evaluated before they are used. In other words, remote access is not statically scheduled.

Johnson [Johnson 1983; Johnson and Fischer 1985] defines “nonlocal attribute grammars” with similar features to remote attribute grammars. Johnson also describes an algo-

rithm for producing a classical (local) attribute grammar from a nonlocal one with user-written annotations. The algorithm includes the same intuition as “fiberizing,” the introduction of attributes that flow along the tree path between remotely connected nodes. Since the nonlocal dependencies are established inside opaque semantic functions, the construction requires the annotation of attributes that carry the information. The example nonlocal attribute grammar for both the dissertation and the later POPL paper establishes a one-to-one connection between declarations and uses, in effect requiring every variable of a block to be mentioned exactly once in the body of the block. This property permits the theory to be symmetric with respect to the definition and the use, but in realistic programming languages, definitions are associated with zero or more uses. The dissertation mentions this possibility, but does not specify how the theory should handle this case. In particular, the problem of multiple nonlocal definitions of an attribute is never addressed, although it appears that some form of a “collection” concept would suffice. Any nontrivial use of collections would, however, immediately run into the problem of “node-wise circularity” and then Johnson and Fischer’s optimal scheduling algorithms would no longer be applicable.

Composable attribute grammars [Farrow et al. 1992] and their extension, composable tree attributions [Boyland and Graham 1994] 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 “nonseparable.” The published references of these two systems require descriptorial composition for implementation. The idea of fiber approximation of this article was based on an incompletely described analysis by Farrow for handling these situations [Farrow 1990].

Vorthmann [1990] uses dynamic scheduling for all attributes in a tree with pre-existing Declaration-Reference (DR) threads, which permit attributes to flow from the declaration to the reference. There is no equivalent to remote field writes. Poetzsch-Heffter [1997] describes the MAX prototyping system based on evolving algebras, with affinities to attribute grammars. Attributes are simply functions and can have syntax-tree nodes as their values; thus one may ask for the attribute of a node computed using another attribute (equivalent to remote attribute reads). The paper does not attempt to check circularity of a specification statically in cases where it uses features beyond classical attribute grammars.

Higher-order attribute grammars [Vogt et al. 1989; Swierstra and Vogt 1991; Saraiva 1999] provide power incomparable with remote attribute grammars: rules may build structured data, but the resulting structure depends on every part of that structure. The structures do not have identity and thus remote attribution is impossible. On the other hand, a computed tree may be rooted in an existing tree and inherited attributes may be provided. Higher-order attribution is particularly useful when modeling translation.

Colander 2 [Maddox 1997] provides the ability to create objects in an attribute grammar-like formalism. An object reference depends on the values of all its fields but cyclic dependencies that arise from the creation of cyclic structure are removed in the same way that cyclic dependencies between fiber approximations are removed, by forcing the evaluation to take two passes. In fact the cycle breaking described in this article was borrowed from Maddox. Automatic cycle breaking avoids the need for loopholes such as in the Elegant system [Augusteijn 1990]. Unlike higher-order attribute grammars, objects may not be rooted and attributed in the tree, but a similar effect is available through the use of

function-valued fields or “methods.” This extension would be attractive for remote attribute grammars as well.

Hedin [1994] defines an extension called door attribute grammars. 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 field writes in power. Again, manual decisions are needed to ensure correct implementation.

Hedin’s later work in “reference attributed grammars” [Hedin 2000] are like our remote attribute grammars, except for collections (remote writes). Reference attributed grammars are implemented automatically, using demand scheduling. Currently there is no incremental evaluation.

Recently, two groups (Magnusson and Hedin [2003] as well as Sasaki and Sassa [2003]) have independently published techniques for evaluating “circular remote/reference attribute grammars” (CRAGs). In both cases, circular dependencies are resolved by repeated evaluation to a least fixed-point. In earlier work [Boyland 1996b] we briefly described a working implementation of CRAGs using demand-driven evaluation and caching of completely evaluated attributes. Magnusson and Hedin use a similar technique and describe how iteration can be avoided when a potential cycle does not occur. Sasaki and Sassa use mostly static scheduling and assume the remote links are in place before evaluation starts, like Vorthmann but unlike our work and that of Magnusson and Hedin. None of these proposals handle circular remote collection attributes. Such attributes were implemented in our earlier work but required trusted annotations to be practical.

## 8. Conclusion

This article presents a formal definition of “remote attribute grammars,” in which dependencies between far-flung portions of a tree may be mediated through fields of objects rather than through the least common ancestor node. If objects may be read from other objects, we find that statically determining circularity is undecidable. However, a family of classes of definitely noncircular remote attribute grammars is defined using approximations of the dependencies. These techniques may be used to schedule practical examples.

## ACKNOWLEDGMENTS

This work was carried out over a ten-year period. It was initially given impetus by an unpublished manuscript sent by Rodney Farrow to the author and Bill Maddox as we were both attempting to implement remote attribute grammars. Conversations with Bill were very helpful. I thank the program committees and participants of CC’98 and LDTA’02 for fruitful feedback. I particularly thank Görel Hedin and João Saraiva for taking time to discuss the issues with me in person. Rong-Qing Tu helped with the implementation of the algorithms. I also thank the anonymous reviewers of this article for many helpful comments.

## REFERENCES

- AIKEN, A. 1996. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices* 31, 7 (July), 19–26.

- ALBLAS, H. 1991. Attribute evaluation methods. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings* (Prague, Czechoslovakia, June 4–13), H. Alblas and B. Melichar, Eds. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, Berlin, 48–113.
- AUGUSTEIJN, L. 1990. The Elegant compiler generator system. In *Attribute Grammars and their Applications. International Conference WAGA Proceedings* (Paris, France, Sept. 19–21), P. Deransart and M. Jourdan, Eds. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, Berlin, 238–254.
- BOCHMANN, G. V. 1976. Semantic evaluation from left to right. *Communications of the ACM* 19, 2 (Feb.), 55–62.
- BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. 1988. CENTAUR: The system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York, 14–24. Appeared as *ACM SIGPLAN Notices*, 24 (2), February 1989.
- BOYLAND, J. 2002. Incremental evaluators for remote attribute grammars. *Electronic Notes in Theoretical Computer Science* 63, 3 (June).
- BOYLAND, J. AND GRAHAM, S. L. 1994. Composing tree attributions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Portland, Oregon, USA). ACM Press, New York, 375–388.
- BOYLAND, J. T. 1996a. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems* 18, 1 (Jan.), 73–108.
- BOYLAND, J. T. 1996b. Descriptive composition of compiler components. Ph.D. thesis, University of California, Berkeley. Available as technical report UCB//CSD-96-916.
- BOYLAND, J. T. 1998. Analyzing direct non-local dependencies in attribute grammars. In *Compiler Construction: 7th International Conference, CC'98* (Lisbon, Portugal, Mar. 28–Apr. 4), K. Koskimies, Ed. Lecture Notes in Computer Science, vol. 1383. Springer, Berlin, Heidelberg, New York, 31–49.
- COURCELLE, B. AND FRANCHI-ZANNETTACCI, P. 1982. Attribute grammars and recursive program schemes. *Theoretical Computer Science* 17, 2–3 (Feb.–Mar.), 163–191, 235–257.
- FARROW, R. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction* (Palo Alto, California, USA). *ACM SIGPLAN Notices* 21, 7 (July), 85–98.
- FARROW, R. 1989. The Linguist translator-writing system. Tech. rep., Declarative Systems Inc. June.
- FARROW, R. 1990. Fibered evaluation in Linguist. unpublished.
- FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. 1992. 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*. 223–234.
- FARROW, R. AND STANCULESCU, A. G. 1989. A VHDL compiler based on attribute grammar methodology. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA). *ACM SIGPLAN Notices* 24, 7 (July), 120–130.
- GRAY, R. W., HEURING, V. P., LEVI, S. P., SLOANE, A. M., AND WAITE, W. M. 1992. Eli: A complete flexible compiler construction system. *Communications of the ACM* 35, 2 (Feb.), 121–131.
- GROSCHE, J. 1990. Object-oriented attribute grammars. In *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)* (Eds. A. E. Harmanci, E. Gelenbe) (Cappadocia, Nevsehir, Turkey). 807–816.
- HEDIN, G. 1994. An overview of door attribute grammars. In *Proceedings of Compiler Construction, 5th International Conference, CC'94* (Edinburgh, U.K.), P. A. Fritzon, Ed. Lecture Notes in Computer Science, vol. 786. Springer-Verlag, Berlin, 31–51.
- HEDIN, G. 2000. Reference attributed grammars. *Informatica* 24, 3, 301–317.
- HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, USA.
- JAZAYERI, M., OGDEN, W., AND ROUNDS, W. 1975. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM* 18, 12 (Dec.), 697–706.
- JOHNSON, G. F. 1983. An approach to incremental semantics. Ph.D. thesis, University of Wisconsin–Madison.

- JOHNSON, G. F. AND FISCHER, C. N. 1985. 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* (New Orleans, Louisiana, USA). ACM Press, New York, 141–151.
- JOURDAN, M., BELLEC, C. L., AND PARIGOT, D. 1990. The OLGA attribute grammar description language: Design, implementation and evaluation. In *Attribute Grammars and their Applications. International Conference WAGA Proceedings* (Paris, France, Sept. 19–21), P. Deransart and M. Jourdan, Eds. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, Berlin, 222–237.
- JOURDAN, M. AND PARIGOT, D. 1991. Internals and externals of the FNC-2 attribute grammar system. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings* (Prague, Czechoslovakia, June 4–13), H. Alblas and B. Melichar, Eds. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, Berlin, 485–504.
- KASTENS, U. 1980. Ordered attributed grammars. *Acta Informatica* 13, 3 (Mar.), 229–256.
- KASTENS, U. 1991. Attribute grammars in a compiler construction environment. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings* (Prague, Czechoslovakia, June 4–13), H. Alblas and B. Melichar, Eds. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, Berlin, 380–400.
- KASTENS, U. AND WAITE, W. M. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31, 601–627.
- KNUTH, D. E. 1968. Semantics of context free languages. *Mathematical Systems Theory* 2, 2 (June), 127–145. Corrections appear in *Mathematical Systems Theory* 5, 1 (1971), 95–96.
- MADDOX, W. 1997. Incremental static semantic analysis. Ph.D. thesis, University of California, Berkeley.
- MAGNUSSON, E. AND HEDIN, G. 2003. Circular reference attributed grammars—their evaluation and applications. In *Proceedings of the Third Workshop on Language Descriptions, Tools and Applications (LDTA 2003). Satellite event of ETAPS 2003* (Warsaw, Poland, Apr. 6). [http://www.di.uminho.pt/~jas/LDTA\\_PROCEEDINGS.tgz](http://www.di.uminho.pt/~jas/LDTA_PROCEEDINGS.tgz).
- PARIGOT, D., ROUSSEL, G., JOURDAN, M., AND DURIS, E. 1996. Dynamic attribute grammars. In *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, H. Kuchen and S. D. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer-Verlag, Berlin, 122–136.
- POETZSCH-HEFFTER, A. 1997. Prototyping realistic programming languages based on formal specifications. *Acta Informatica* 34, 737–772.
- REPS, T. 1998. Program analysis via graph reachability. Tech. Rep. CS-TR-1998-1386, University of Wisconsin, Madison.
- REPS, T. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan.), 162–186.
- REPS, T. W. AND TEITELBAUM, T. 1989. *The Synthesizer Generator: A system for constructing language-based editors*. Springer-Verlag, Berlin.
- RODEH, M. AND SAGIV, M. 1999. Finding circular attributes in attribute grammars. *Journal of the ACM* 46, 4 (July), 556–575.
- SARAIVA, J. 1999. Purely functional implementation of attribute grammars. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands. <ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/>.
- SASAKI, A. AND SASSA, M. 2003. Circular attribute grammars with remote attribute references and their evaluators. *New Generation Computing* 22.
- SWIERSTRA, D. AND VOGT, H. 1991. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings*, H. Alblas and B. Melichar, Eds. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, Berlin, 256–296.
- UHL, J., DROSSOPOULOU, S., PERSCH, G., GOOS, G., DAUSMANN, M., WINTERSTEIN, G., AND KIRCHGÄSSNER, W. 1982. *An Attribute Grammar for the Semantic Analysis of Ada*. Lecture Notes in Computer Science, vol. 139. Springer-Verlag, Berlin.
- VOGT, H. H., SWIERSTRA, S. D., AND KUIPER, M. F. 1989. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA). *ACM SIGPLAN Notices* 24, 7 (July), 131–145.

VORTHMANN, S. A. 1990. Coordinated incremental attribute evaluation on a DR-threaded tree. In *Attribute Grammars and their Applications. International Conference WAGA Proceedings* (Paris, France, Sept. 19–21), P. Deransart and M. Jourdan, Eds. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, Berlin, 207–221.

RECEIVED NOVEMBER 2003; REVISED NOVEMBER 2004; ACCEPTED DECEMBER 2004.