INFORMATION AND COMPUTATION 78, 1-55 (1988)

Proofs of Partial Correctness for Attribute Grammars with Applications to Recursive Procedures and Logic Programming[*,†]

B. COURCELLE

Université Bordeaux I, Département d'Informatique, 351, Cours de la Libération, 33405 Talence, France

AND

P. DERANSART

INRIA, Domaine de Voluceau, Rocquencourt, B. P. 105, 78153 Le Chesnay Cédex, France

An extension of the inductive assertion method allowing one to prove the partial correctness of an attribute grammar w.r.t. a specification is presented. It is complete in an abstract sense. It is also shown that the semantics of systems of recursive imperative procedures or of recursive applicative procedures computed with call-by-value or call-by-name can be expressed by an attribute grammar associating attributes with the nodes of the so-called trees of calls. Hence the proof methods for the partial correctness of attribute grammars can be applied to these recursive procedures. We show also how the proof method can be applied in logic programming. © 1988 Academic Press, Inc.

INTRODUCTION

The problem of proving the validity of an attribute grammar with respect to a given specification, saying what the values of the attributes should be, although essential, has been rarely considered (Pair *et al.*, 1979; Katayama and Hoshino, 1981) and (Deransart, 1983; Courcelle, 1984) are preliminary versions of the present work. Since an attribute grammar, designed say, for specifying a compiler, can be very large, proving its validity, i.e., the correctness of the generated compiler, is clearly not a trivial task.

* This work has been supported by the "Programme de Recherches Coordonées: Programmation avancée et outils pour l'intelligence artificielle."

† email (UUCP network): courcell@geocub.inria.fr or deransar@minos.inria.fr.

We shall consider here the *partial correctness* of an attribute grammar with respect to a *specification.* A specification consists in a logical formula associated with each non-terminal and establishing a relation between the values of the attributes at a node labelled by this non-terminal. An attribute grammar is *partially correct* if, *for every tree t*, for every assignment of values to the attributes at the nodes of the tree t which satisfies the semantic rules, the specification is satisfied at the root of the tree (there is no distinguished start symbol in this approach). In terms of flowcharts, this corresponds to saying that *for every computation path*, the output values satisfy some prescribed relation with the input ones.

The *total correctness* of an attribute grammar requires the existence of *at least one tree*, with some assignment of values to attributes satisfying some condition. This problem will not be considered here (but it has been in (Deransart, 1984)).

We propose two proof methods for establishing the partial correctness of an attribute grammar. The first consists in defining a specification stronger than the original one, which furthermore is *inductive.* This means that for each node of a derivation tree, the validity of the specification at this node follows from the validity of the specification at its sons and the semantic rules of the production associated with this node.

Since the strongest specification is inductive this proof method is complete, provided one accepts for specifications arbitrary set theoretical relations as opposed to, say, first-order formulas. Otherwise, one gets an incompleteness result similar to Wand's for the inductive assertion method (Wand, 1978).

Our second method is a refinement of the first. We associate with every non-terminal a finite set of formulas (we call this an *annotation*), together with implications between formulas (and using the semantic rules as premises). Some formulas are called *inherited*, others *synthesized.* They behave with respect to implications as inherited and synthesized attributes behave w.r.t. dependencies, and we require non-circularity exactly as we do for ordinary attribute grammars. This non-circularity ensures the non-circularity of the proof that, for every tree, if the inherited formulas are valid at the root then so are the synthesized ones.

Hence, in order to prove the validity of a specification $\Theta = \{\Theta^X\}_{X \in N}$, it suffices to find an annotation which is non-circular and such that, for every non-terminal X,

$$(\mathbb{AND}\ \mathscr{I}(X) \Rightarrow \mathbb{AND}\ \mathscr{S}(X)) \Rightarrow \Theta^X,$$

where $\mathscr{I}(X)$ (resp. $\mathscr{S}(X)$) is the set of inherited (resp. synthesized) formulas at the non-terminal X (and $\mathbb{AND}$ denotes the conjunction of a set of formulas).

The first method is a special case of the second since it corresponds to taking one synthesized formula for each non-terminal (and the non-circularity is then trivial as for purely synthesized attribute grammars). Roughly speaking, the second one consists in decomposing specifications into implications between conjuncts of simpler formulas; it is useful in practice to get clearer proofs as we shall show with examples. It is shown in Section 5 that they are equally powerful.

The originality of our approach is that our proof methods do not rest on the non-circularity of the given attribute grammar as did those of (Pair, 1979) or (Katayama, 1981). (Recall that the partial correctness is stated as "for every tree, for every assignment satisfying the semantic rules..."; there may exist none or many in the case of a circular attribute grammar.) Hence we shall formulate our methods for a generalization of attribute grammars called *relational attribute grammars* where the semantic rules do not state that some attribute *is a function of other ones* but simply state *relations* (possibly not functional in any way) between attributes. Hence, there is no distinction between synthesized and inherited attributes. Clearly non-circularity is meaningless for relational attribute grammars.

In our proof method, we need a non-circularity at the level of logical formulas, not at the level of the attribute grammar we are validating.

The proof method using an inductive specification is reminiscent of the classical *fix-point induction* for recursive procedures.

We strengthen this analogy by showing that the semantics of certain recursive procedures either imperative or applicative can be formalized in terms of conditional attribute grammars (a conditional attribute grammar is an ordinary attribute grammar where a boolean condition is associated with each production; hence some trees may have no valid assignment of values to the attributes of their nodes even if the grammar is non-circular).

The class of conditional attribute grammars lies between the class of attribute grammars and the class of relational attribute grammars. It has already been introduced in Watt and Madsen (1983), but here we give another illustration of this concept.

The conditional attribute grammar associated with a recursive procedure is based on the set of *trees of calls* of the recursive procedure. Each node corresponds to "a call" (the root corresponds to the "main program"), is labelled by the name of the procedure which is called, and has attributes representing the parameters: the inherited attributes represent the input parameters and the synthesized attributes the output parameters. Each "production rule" corresponds to some alternative in the procedure body which is selected by some boolean test. The semantic rules of such a "production" include the test together with operations performed on the program variables (which are represented by attributes). The underlying attribute grammar is of a rather simple kind: it is an *L*-attribute grammar, i.e., one for which the attributes are evaluable in one left-to-right pass (Engelfriet, 1984)).

This attribute grammar is used in an unusual way: the tree is not given a priori and then decorated and evaluated, but is constructed in interaction with the computation of attributes; this construction is oriented by the boolean conditions which determine it completely if the procedures are deterministic or only forbid some alternatives if they are non-deterministic.

By means of this construction the partial correctness of recursive procedures can be expressed as the partial correctness of the associated attribute grammar, and the proof methods for attribute grammars are applicable. But this does not yield any new result.

We also apply this method to the correctness of definite clause programs, i.e., the correctness of "pure" PROLOG programs. By correctness of a definite clause program w.r.t. a specification $\mathcal{S}$ we mean that all possible answer substitutions satisfy the specification $\mathcal{S}$. It has been shown in (Deransart and Maluszynski, 1985) how definite clause programs can be considered as relational attribute grammars. Hence our proof methods give rise directly to validation methods for logic programming. See Deransart and Ferrand (1987).

This paper is organized as follows. Section 1 consists of definitions concerning many-sorted algebras and attribute grammars. Section 2 shows the use of attribute grammars for studying the semantics of recursive procedures and logic programs. Section 3 deals with fix-point induction for attribute grammars (i.e., with the first proof method mentioned above). Section 4 introduces annotations (the second method) and Section 5 establishes that these two methods are equally powerful.

(1) Basic Definitions and Notations

Attribute grammars will be defined in the algebraic style of Chirica and Martin (1979) or Courcelle and Franchi-Zannettacci (1982).

(1.1) *Sorts, Signatures, Terms*

Let S be a finite set of *sorts*. An S-sorted *signature* (or simply an *S-signature*) is a finite set P of function symbols given with two mappings:

$$\alpha: P \to S^* \qquad (\alpha(p) \text{ is called the } \textit{arity} \text{ of } p \text{ in } P)$$

$$\sigma: P \to S \qquad (\sigma(p) \text{ is called the } \textit{sort} \text{ of } p \text{ in } P).$$

The length of $\alpha(p)$ is called the *rank* of p and is denoted by $\rho(p)$. If $\alpha(p) = \varepsilon$ (we denote by ε the empty word of any free monoid except in one case (see (1.2) below), then p is a constant symbol. The pair $\langle \alpha(p), \sigma(p) \rangle$

is the *profile* of p. A constant (or a variable) has profile $\langle \varepsilon, s\rangle$. $P_{\langle w, s\rangle}$ denotes the subset of P consisting of functions of profile $\langle w, s\rangle$. The notation p: $\langle w, s\rangle$ is a shorthand for $p \in P_{\langle w,s\rangle}$.

A *heterogeneous P-magma* (or *-algebra*) is an object $\mathbb{M}$,

$$\mathbb{M} = \langle \{M_s\}_{s\in S}, \{p_{\mathbb{M}}\}_{p\in P}\rangle,$$

where $\{M_s\}$ is a family of sets indexed by S, the *carriers*, and each $p_{\mathbb{M}}$ is a mapping,

$$M_{s1} \times \cdots \times M_{sn} \to M_s \qquad \text{if} \quad p \in P_{\langle s1,\ldots,sn,s\rangle}.$$

Unless otherwise specified, all magmas considered in the sequel will be heterogeneous.

Let X be an S-sorted set of variables (i.e., each x in X has arity ε and a sort $\sigma(x)$ in S, and $X_s = \{x \in X/\sigma(x) = s\}$); one can also define the *free-P-magma* generated by X, denoted by $M(P, X)$. It is identified as usual with a set of terms, "well typed" with respect to sorts and arities. Terms will be written with commas and parentheses if infix notation is not used. They will also be identified with trees in a well-known manner. The words "tree" and "term" will be synonymous in this paper and will refer to elements of some free magma. We shall denote by $M(P, X)_s$ the carrier of sort s of $M(P, X)$, and by $M(P)_s$ the set of all terms without variables (*ground* terms) of sort s. $M(P)$ is the set of all ground terms.

A term t in $M(P)_s$ is considered as denoting a value $t_{\mathbb{M}}$ in M_s for a P-magma $\mathbb{M}$. Similarly a term t in $M(P, \{x_1, \ldots, x_k\})_s$ is considered as denoting a function,

$$t_{\mathbb{M}}\colon M_{\sigma(x_1)} \times \cdots \times M_{\sigma(x_k)} \to M_s, \qquad \text{called a } \textit{derived operator}.$$

For any P-magma $\mathbb{M}$ and S-sorted set X of variables, an *assignment* of values to variables is an S-indexed family of functions:

$$v = \{v_s\colon X_s \to M_s\}_{s\in S} \qquad \text{(also denoted by } v\colon X \to M).$$

It is well known that this assignment can be extended into a unique homomorphism v': $M(P, X) \to \mathbb{M}$ such that $v'(x) = v(x)$ for all x in X.

Let Q be an S-indexed family of propositions $\{Q_s\}_{s\in S}$, where Q_s is a unary proposition on $M(P)_s$. We say that the proposition

(1) $\forall s \in S, \forall t \in M(P)_s, Q_s(t)$

is *provable by structural induction* iff conditions (2) and (3) below are both provable:

(2) $\forall s \in S, \forall p \in P_{\langle \varepsilon,s\rangle}, Q_s(p)$

(3) $\forall s_1, \ldots, s_n,\ s \in S,\ \forall p \in P_{\langle s_1,\ldots,s_n,s\rangle}\ \forall t_1 \in M(P)_{s_1}, \ldots,\ \forall t_n \in M(P)_{s_n}$: $Q_{s_1}(t_1)$ **and** $\cdots$ **and** $Q_{s_n}(t_n) \Rightarrow Q_s(p(t_1, \ldots, t_n))$.

It is easy to show that a proposition is true if it is provable by structural induction (an elegant proof is given in (Chirica and Martin, 1979)).

(1.2) *Trees and Grammars*

Let P be a fixed S-signature. By a *tree* we mean an element of $M(P)$ or of $M(P, X)$. Let $n_P = \mathrm{Max}\{\rho(p)/p \in P\}$.

For any integer n, we let $[n]$ denote the set $\{1, 2, ..., n\}$ if $n \geqslant 1$ and $\varnothing$ if $n = 0$.

The nodes of a tree t will be represented in Dewey notation by words in $[n_P]^*$. Whereas we shall use ε to denote the empty word of any free monoid, we shall use 0 for the empty word of the free monoid $[n_P]^*$, just to simplify some notations. We denote by $\mathrm{Node}(t)$ the set of nodes of a tree t; hence $\mathrm{Node}(t) \subseteq [n_P]^*$ and $0 \in \mathrm{Node}(t)$ (0 denotes the root of the tree t).

Every node u in t is labelled by a unique element p of P denoted by $\mathrm{lab}_t(u)$. We define the sort of u (and denote it by $\sigma_t(u)$, or $\sigma(u)$ if t is clear) as the sort of $\mathrm{lab}_t(u)$ and the sort of t (denoted by $\sigma(t)$) as the sort $\sigma_t(0)$ of its root.

Let u be a node of t. We denote by t/u the subtree of t issued from u.

We denote by $t[t_1/x_1, ..., t_k/x_k]$ the result of the simultaneous substitution in t of t_1 for $x_1, ..., t_k$ for x_k, where $x_1, ..., x_k$ are pairwise distinct variables.

If $P \supseteq Q$ and $M(P, X) \supseteq T$ we denote by $Q(T)$ the set of trees in $M(P, X)$ of the form $q(t_1, ..., t_k)$ with $q \in Q$ and $t_1, ..., t_k \in T$. We denote by $M(P, T)$ the set of trees of the form $t[t_1/x_1, ..., t_k/x_k]$ for t in $M(P, \{x_1, ..., x_k\})$ and $t_1, ..., t_k \in T$.

An attribute grammar can be thought of as a way to associate a meaning with a *derivation tree* (i.e., an abstract syntax tree) rather than with a word, and this approach eliminates the need for a non-ambigous grammar.

In order to emphasize this fact we redefine as follows a context-free grammar. A *context-free grammar* is a triple $\langle N, P, C\rangle$ where N is a finite set (the "non-terminal alphabet"), P is a finite N-signature, and C is a mapping associating with every p in P of profile $\langle X_1 \cdots X_n, X_0\rangle$ an $(n+1)-$tuple of words on some finite alphabet T (the "terminal alphabet").

An *abstract context free grammar* is a pair $\langle N, P\rangle$ as above. A production rule $X_0 \to \alpha_1 X_1 \alpha_2 X_2 \cdots \alpha_n X_n \alpha_{n+1}$ in the usual sense is split here into a symbol p, its name with profile $\langle X_1 \cdots X_n, X_0\rangle$, and the $(n+1)$-tuple $C(p) = \langle \alpha_1, ..., \alpha_{n+1}\rangle$ ($\alpha_1, ..., \alpha_{n+1}$ are words in T^*).

(1.3) *Many-Sorted Logical Languages*

Let S be a finite set of sorts. Let $\mathbf{b}$ denote an additional sort, that of the boolean values **true** and **false**.

Let $\vartheta = \{\vartheta_s\}_{s \in S}$ be a sorted set of variables, let $\mathscr{F}$ be an S-signature, and let $\mathcal{R}$ be a finite set of many-sorted predicate symbols (i.e., a set of symbols $\mathcal{R}$, each $R \in \mathcal{R}$ having an arity $\alpha(R) \in S^+$ and, implicitly, the sort **b**).

A *logical language* over $(\vartheta, \mathcal{F}, \mathcal{R})$ consists of a set of *formulas* $\mathcal{L}$ written with ϑ, $\mathcal{F}$, $\mathcal{R}$ and connectives like $\forall$, $\exists$, $\Rightarrow$. We do not give any more detail here. We assume that each formula φ in $\mathcal{L}$ has a (possibly empty) set of free variables Free (φ). For $\vartheta \supseteq \vartheta'$ we denote by $\mathcal{L}(\vartheta')$ the logical language $\{\varphi \in \mathcal{L} / \vartheta' \supseteq \text{Free}(\varphi)\}$.

Let $\mathcal{C}$ denote a class of *structures*, i.e., of objects, of the form $\mathbb{D} = \langle (D_s)_{s \in S}, (f_{\mathbb{D}})_{f \in \mathcal{F}}, (r_{\mathbb{D}})_{r \in \mathcal{R}} \rangle$ where $\langle (D_s)_{s \in S}, (f_{\mathbb{D}})_{f \in \mathcal{F}} \rangle$ is a heterogeneous $\mathcal{F}$-magma and for each r in $\mathcal{R}$, $r_{\mathbb{D}}$ is a total mapping $D_{s_1} \times \cdots \times D_{s_n} \to \{\textbf{true}, \textbf{false}\}$ where $\alpha(r) = s_1 \cdots s_n$, i.e., in other words an n-ary relation on $D_{s_1} \times \cdots \times D_{s_n}$.

Let $\models$ denote the *validity* relation defined as follows. For every assignment ν as in (1.1), every $\mathbb{D}$ in $\mathcal{C}$, every φ in $\mathcal{L}$ one assumes that $(\mathbb{D}, \nu) \models \varphi$ either holds or does not hold. In the former case we also say that φ holds in $(\mathbb{D}, \nu)$ and in the latter that it does not.

We write $\mathbb{D} \models \varphi$ if $(\mathbb{D}, \nu) \models \varphi$ holds for every assignment ν. In many cases we shall identify $\mathcal{L}$ with the triple $\langle \mathcal{L}, \mathcal{C}, \models \rangle$ if no confusion can arise.

The logical connectors we shall use are **and**, **or**, **not**, $\Rightarrow$, $\Leftrightarrow$. For a finite set of formulas A, we denote by $\mathbb{AND}\, A$ (resp. $\mathbb{OR}\, A$) the conjunction (resp. the disjunction) of A (with $\mathbb{AND}\, \varnothing = \textbf{true}$ and $\mathbb{OR}\, \varnothing = \textbf{false}$). Finally, if $\Phi \in \mathcal{L}(\{v_1, ..., v_k\})$, we denote by $\Phi[w_1/v_1, ..., w_k/v_k]$ (or $\Phi[w_i/v_i;\ 1 \leqslant i \leqslant k]$) the result of the substitution of w_i for each free occurrence of v_i (some renaming of variables may be necessary).

(1.4) *Relational Attribute Grammars*

(1.4.1) Definition. A *relational attribute grammar* is a 5-tuple $G = \langle N, P, \text{Attr}, \Phi, \mathbb{D} \rangle$ consisting of

(1) an abstract context free grammar $\langle N, P \rangle$

(2) a set of attributes Attr defined as

(i) a finite set Attr of attributes such that $\text{Attr} = \bigcup_{X \in N} \text{Attr}(X)$ where $\text{Attr}(X)$ is the set of attributes of X in N (one can have $\text{Attr}(X) \cap \text{Attr}(X') \neq \varnothing$, $X \neq X'$).

(ii) every attribute a in Attr has a sort $\sigma(a)$ in the set of sorts S of $\mathcal{L}$.

(3) a set of relations Φ defined as

(iii) a logical language $\langle \mathcal{L}, \mathcal{C}, \models \rangle$

(iv) for each production p of profile $\langle X_1 \cdots X_n, X_0 \rangle$ a formula Φ_p belonging to the logical language $\mathcal{L}(W(p))$ where $W(p)$ is now defined.

For each i in $\{0, ..., n\}$ and each a in $\mathrm{Attr}(X_i)$ we introduce a new symbol $a(i)$ and call it an *occurrence of the attribute a* in p. We denote by $W(p)$ the set of these symbols. We also say that $W(p)$ is the set of *attribute occurrences* of p.

The sort $\sigma(a(i))$ is defined as $\sigma(a)$.

(4) $\mathbb{D}$ is a structure in $\mathscr{C}$.

Convention. The sets $\mathrm{Attr}(X)$ will be ordered in a fixed way so that $\mathrm{Attr}(X)$ will be used as a sequence (an element of Attr^*) in some cases.

(1.4.2) *Semantics of a relational attribute grammar.* Let $t \in M(P)$. We define a set of variables $W(t)$ called *the set of attribute occurrences* of t as

$$W(t) = \{a(u)/u \in \mathrm{Node}(t), a \in \mathrm{Attr}(\sigma_t(u))\}.$$

Hence $a(u)$ is a new symbol. The sort of $a(u)$ is defined as $\sigma(a(u)) = \sigma(a)$. We denote by $W_0(t)$ the set $\{a(0)/a \in \mathrm{Attr}(\sigma_t(0))\}$.

For each $u \in \mathrm{Node}(t)$ we denote by $\Phi_u \in \mathscr{L}(W(t))$ the formula

$$\Phi_p[a(ui)/a(i);\ a(i) \in W(p)],$$

where $p = \mathrm{lab}_t(u)$. Recall that 0 denotes the empty word so that $ui = u$ for $i = 0$.

We denote by Φ_t the conjunction of all the Φ_u's, for $u \in \mathrm{Node}(t)$.

An assignment v: $W(t) \to D$ is called a *t-assignment.* It is *valid* if $(\mathbb{D}, v) \models \Phi_t$.

One may be interested in all valid assignments or simply in their restrictions to the set $W_0(t)$ of the attribute occurrences at the root of t (i.e., $W_0(t) = \{a(u) \in W(t)/u = 0\}$). In this case we shall use the relation $R_{t,\mathbb{D}} \subseteq D_{\sigma(a_1)} \times \cdots \times D_{\sigma(a_k)}$ (where $(a_1, ..., a_k)$ is the sequence $\mathrm{Attr}(\sigma_t(0)))$ defined by

$(d_1, ..., d_k) \in R_{t,\mathbb{D}}$ iff there exists a valid t-assignment v such that $v(a_i(0)) = d_i$ for $i = 1, ..., k$.

Note that with these definitions there is no distinction between synthesized and inherited attributes and $R_{t,\mathbb{D}}$ is always defined (but possibly empty) without any extra-condition like non-circularity.

(1.4.3) EXAMPLE. In the following example, attributes are used to compute the longest prefix v of a word $w \in \{a, b\}^+$ with $v \in b^+ \cup b^*a$. Two boolean functions on words are used: **bword**(w) is **true** iff $w \in b^*$, **baword**(w) is **true** iff $w \in b^*a$.

We present two relational attribute grammars which compute the same prefix, the equivalence of the two attribute grammars will be demonstrated later,

G_2: $N = \{S, A, a, b\}$

$P = \{p1\colon \langle A, S\rangle, p2\colon \langle AA, A\rangle, p3\colon \langle a, A\rangle,$
$p4\colon \langle b, A\rangle, p5\colon \langle \varepsilon, a\rangle, p6\colon \langle \varepsilon, b\rangle\}$

$\mathrm{Attr}(S) = \{x\}$

$\mathrm{Attr}(A) = \{u, x\}$

$\mathrm{Attr}(a) = \mathrm{Attr}(b) = \varnothing$

$\mathrm{Inh} = \{u\}$

$\mathrm{Syn} = \{x\}$ u and x are of type word.

G_3: $N =$ as in G_2

$P =$ as in G_2

$\mathrm{Attr} = \mathrm{Attr}(S) = \mathrm{Attr}(A) = \{y\}$

$\mathrm{Inh} = \varnothing$

$\mathrm{Syn} = \{y\}$ y is of type word.

The relations are the implicit conjunctions of the formulas shown in the boxes in Figs. 1 and 2 (hence Φ_{p1} is: "$u(1) = \varepsilon$ **and** $x(0) = x(1)$").

(1.5) *(Usual) Attribute Grammars*

An *attribute grammar* is a relational attribute grammar $\langle N, P, \mathrm{Attr}, \Phi, \mathbb{D}\rangle$ such that $\mathrm{Attr} = \mathrm{Inh} \cup \mathrm{Syn}$ is the union of two disjoint sets (inherited and synthesized attributes). We denote $\mathrm{Inh}(X) = \mathrm{Attr}(X) \cap \mathrm{Inh}$ and $\mathrm{Syn}(X) = \mathrm{Attr}(X) \cap \mathrm{Syn}$.

We partition each $W(p)$ as follows.

$W(p) = W_{\mathrm{in}}(p) \cup W_{\mathrm{out}}(p)$ is the disjoint union of the *input* and *output* attribute occurrences of p in $P_{\langle X_1,\ldots,X_n,X_0\rangle}$ defined as

$$W_{\mathrm{in}}(p) = \{a(i)/a \in \mathrm{Inh}(X_0) \text{ and } i = 0, \text{ or } a \in \mathrm{Syn}(X_i) \text{ and } 1 \leqslant i \leqslant n\}$$

$$W_{\mathrm{out}}(p) = \{a(i)/a \in \mathrm{Syn}(X_0) \text{ and } i = 0, \text{ or } a \in \mathrm{Inh}(X_i) \text{ and } 1 \leqslant i \leqslant n\}.$$

The formulas Φ_p satisfy the following (where $\mathscr{F}$ is the S-signature of $\mathscr{L}$). For each attribute occurrence w of $W_{\mathrm{out}}(p)$ one has a formula $\Phi_{p,w}$ of the form $w = t$, where $t \in M(\mathscr{F}, W(p))$ and Φ_p is:

$$\mathbb{AND}\{\Phi_{p,w}/w \in W_{\mathrm{out}}(p)\}.$$

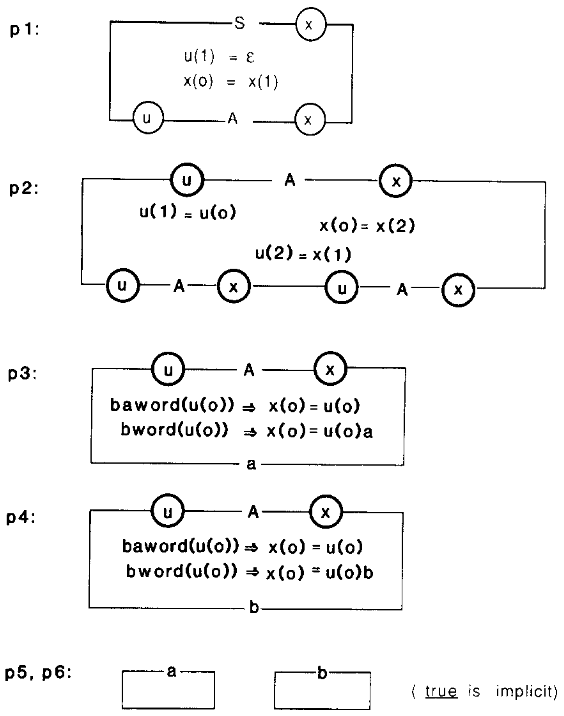

FIGURE 1

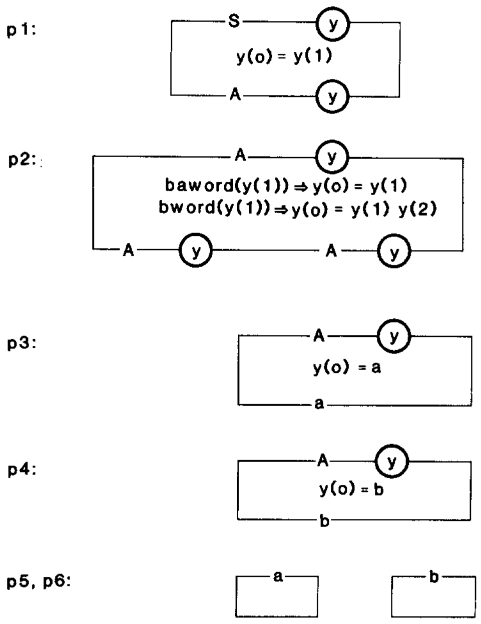

FIGURE 2

The formulas $\Phi_{p,w}$ are *attribute definitions.* We refer to Φ_p as to the *semantic rules* of production *p.* If all the terms *t* as above are in $M(\mathscr{F}, W_{\text{in}}(p))$ we say that *G* is in *normal form.*

(1.5.1) Example. We borrow from Katayama and Hoshino (1981) the following school example which serves to compute four times the height of a linear tree,

$$
\begin{aligned}
G_1\colon\ & N = \{S, A\} \\
& P = \{p_1, p_2, p_3\} \qquad \text{with} \quad p_1\colon \langle A, S\rangle,\ p_2\colon \langle A, A\rangle,\ p_3\colon \langle \varepsilon, A\rangle \\
& \mathrm{Attr} = \mathrm{Attr}(S) \cup \mathrm{Attr}(A) \\
& \mathrm{Attr}(S) = \{k\} \\
& \mathrm{Attr}(A) = \{f, h, g, k\} \\
& \mathrm{Inh} = \{f, h\} \\
& \mathrm{Syn} = \{g, k\}.
\end{aligned}
$$

Attribute definitions are written inside squares in Fig. 3 recalling the gram-

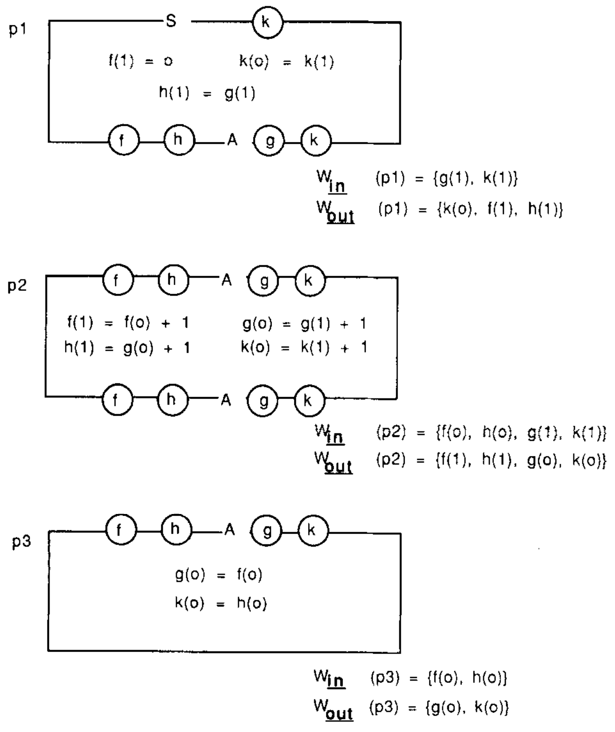

Figure 3

matical rule profile. Non-terminal (or set) symbols are numbered from left to right. Hence Φ_{p1} is: "$f(1)=0$ **and** $k(0)=k(1)$ **and** $h(1)=g(1)$." Note that it is a usual attribute grammar, since all attributes in $W_{\text{out}}(p_i)$ have a functional definition.

(1.6) *Conditional Attribute Grammars*

A *conditional attribute grammar* is similar to an attribute grammar. The only difference concerns the formulas Φ_p which are of the form

$$B_p \ \textbf{and}\ \mathbb{AND}\{\Phi_{p,w}/w \in W_{\text{out}}(p)\},$$

where the $\Phi_{p,w}$'s are as in (1.5) and B_p is a boolean conjunction of atomic formulas of the form $R(t_1, ..., t_k)$ or $\textbf{not}(R(t_1, ..., t_k))$ for some k-ary predicate symbol R in $\mathscr{R}$ and some terms $t_1, ..., t_k$ in $M(\mathscr{F}, W_{\text{in}}(p))$. Examples of conditional attribute grammars are given in the next section (see (2.3.4)).

(2) Relation to Recursive Procedures and Logic Programming

(2.1) *Recursive Imperative Procedures*

We show that the semantics of certain recursive imperative procedures can be formalized by means of attributes associated with the nodes of a tree called the *tree of calls* which defines the structure of recursive calls in some computation for some interpretation and some values of the arguments.

A *conditional attribute grammar* will be defined, based on a signature which corresponds to the structure of recursive calls. In order to give precise statements, we need a precise class of programs. We shall use the one introduced by Gallier (1981).

(2.1.1) Definitions. Let $\mathscr{F}$, $\mathscr{R}$, ϑ be as in (1.3). Since the set of sorts S will be irrelevant we shall imply take it reduced to a single sort. The extension to multiple sorts is trivial.

The set ϑ will be partitioned into $\vartheta_{\text{I}} \cup \vartheta_{\text{L}} \cup \vartheta_{\text{O}}$ where $\vartheta_{\text{I}} = \{x_1, x_2, ..., x, x', x'', ...\}$ is the set of *input* variables, $\vartheta_{\text{L}} = \{y_1, y_2, ..., y, y', ...\}$ is the set of *local* variables, and $\vartheta_{\text{O}} = \{z_1, z_2, ..., z, z', ...\}$ is the set of *output* variables.

We shall denote by $\vartheta_{\text{I},k}$ the set $\{x_1, x_2, ..., x_k\}$ and similarly for $\vartheta_{\text{L},k}$, $\vartheta_{\text{O},k}$.

Let $\mathscr{A}$ be a finite set of procedure symbols; each A in $\mathscr{A}$ has a rank $\rho(A) = (n, m)$ which is a pair of positive integers.

A *procedure definition* over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ is an equation

$$A(x_1, ..., x_n; z_1, ..., z_m) = S_A,$$

where $(n, m) = \rho(A)$ and S_A is a flowchart over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ of rank (n, m). We shall abbreviate it to

$$A(\mathbf{x}; \mathbf{z}) = S_A,$$

where $\mathbf{x}$ represents the list $(x_1, ..., x_n)$ called the list of *formal input parameters* and $\mathbf{z}$ represents $(z_1, ..., z_m)$, the list of *formal output parameters.*

A non-deterministic *flowchart* over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ of rank (n, m) consists of:

(1) a finite directed graph G with a special node **in** called the *entry* node, a special node **out** called the *exit* node, and such that **in** (resp. **out**) is not the target (resp. the source) of any edge and such that each node belongs to some path from **in** to **out**, and

(2) a labelling function which associates an instruction over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ of type (n, p, m) for some p with every edge, and satisfying a condition which will be stated later in terms of computation paths.

By an *instruction* of type (n, p, m) over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ we mean

(1) **null**, the null instruction,

(2) or a *simultaneous assignment* of the form

$$(v_1, ..., v_k) \leftarrow (t_1, ..., t_k),$$

where $v_1, ..., v_k$ are pairwise distinct variables in

$$\vartheta_{\mathrm{O},m} \cup \vartheta_{\mathrm{L},p} \quad \text{and} \quad t_1, ..., t_k \in \mathscr{F}(\vartheta_{\mathrm{I},n} \cup \vartheta_{\mathrm{L},p}) \cup \vartheta_{\mathrm{I},n} \cup \vartheta_{\mathrm{L},p}$$

(see (1.2) for notations),

(3) or a *guard*, i.e., an instruction of the form

$$R(v_1, ..., v_k)$$

or

$$\mathbf{not}\ (R(v_1, ..., v_k))$$

for some R in $\mathscr{R}_k$ (the set of predicate symbols of $\mathscr{R}$ of rank k), $v_1, ..., v_k$ in $\vartheta_{\mathrm{I},n} \cup \vartheta_{\mathrm{L},p} \cup \vartheta_{\mathrm{O},m}$,

(4) or a *procedure call*, i.e., an instruction of the form $\mathbf{call}\, A(u_1, ..., u_{n'};\ v_1, ..., v_{m'})$ where $A \in \mathscr{A}$, $\rho(A) = (n', m')$, $u_1, ..., u_{n'} \in \vartheta_{\mathrm{I},n} \cup \vartheta_{\mathrm{L},p}$, $v_1, ..., v_{m'} \in \vartheta_{\mathrm{L},p} \cup \vartheta_{\mathrm{O},m}$ and $v_i \neq v_j$ for $i \neq j$ and $v_i \neq u_j$ for all i, j.

Such a procedure call is abbreviated $\mathbf{call}\, A(\mathbf{u}; \mathbf{v})$; $\mathbf{u}(\mathbf{v})$ is the list of *actual input* (*output*) *parameters.*

A *system of recursive imperative procedures* Σ over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ is a set of procedure definitions over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ such that for every A in $\mathscr{A}$ there is one and only one definition with left-hand side of the form $A(\mathbf{x}; \mathbf{z})$.

A *recursive imperative program scheme over* $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ (we shall simply say a *scheme* in this section), is a pair $\langle \Sigma, A \rangle$ consisting of a system of recursive imperative procedures Σ over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ and some A in $\mathscr{A}$ playing the role of the "main program." The rank of the scheme is defined as that of A.

An *interpretation* for (Σ, A) (or Σ) is any structure $I = \langle D, (f_I)_{f \in \mathscr{F}}, (r_I)_{r \in \mathscr{R}} \rangle$ where the f_I's and the r_I's are total functions and relations over D of appropriate arity (as in (1.3)).

A *recursive imperative program* P is a pair $\langle \langle \Sigma, A \rangle, I \rangle$ consisting of a scheme and an interpretation. Such a program defines a relation $P_I \subseteq D^n \times D^m$ where $\rho(A) = (n, m)$. The definition of P_I is recalled informally in Section (2.2) below. A formal definition can be found in Gallier (1981).

(2.1.2) EXAMPLE. Let Σ be reduced to the single definition $A(x_1, x_2; z) = S$ where S is the flowchart shown in Fig. 4.

In order to help the reader to make a correspondence between Fig. 4 and Definition (2.1.1) we make precise that the nodes of the underlying graph are **in**, α, β, γ, δ, ε, **out**. The instructions labelling the edges are indicated in boxes.

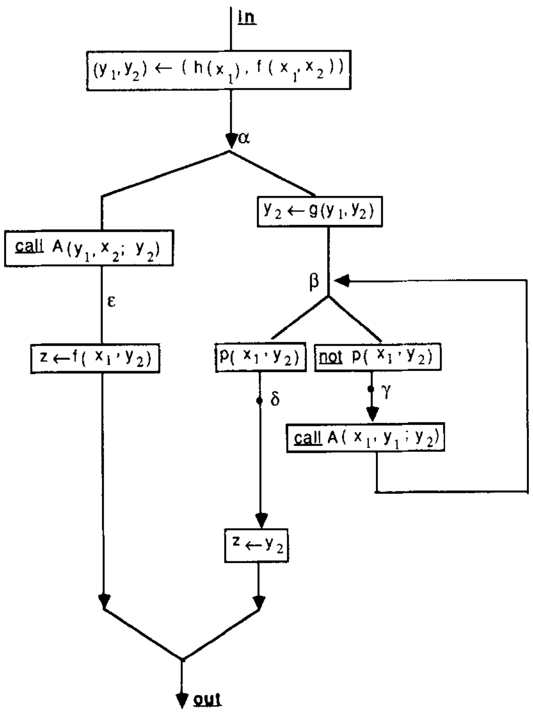

FIGURE 4

The node α is a non-deterministic choice. The node β is a deterministic choice due to the presence of the two exclusive guards $p(x_1, y_2)$ and $\mathbf{not}(p(x_1, y_2))$: at most one of the edges (β, δ) and (β, γ) can be taken in each case.

(2.1.3) *Syntactic restrictions.* The above definitions reproduce Gallier's except that we have omitted his *relational assignments* (just for simplifying the exposition, but all our definitions and results could be extended so as to admit them).

We shall make another more important restriction. A flowchart is *loop-free* if its underlying graph has no cycle. It is well known that a scheme $\langle \Sigma, A \rangle$ where some right-hand sides of procedure definitions are not loop-free can be transformed into an equivalent one with loop-free right-hand sides at the cost of introducing new procedure symbols. A special case of this transformation is the recursive definition of the *while* construct (see Mc Carthy *et al.* (1965)). Our construction only works for *loop-free* systems in a fundamental way.

(2.2) *Operational Semantics*

(2.2.1) *Informal definition.* We recall the definition given by Gallier (1981, pp. 204–209). We first consider the case of a flowchart S of rank (n, m) over $(\mathscr{F}, \mathscr{R}, \varnothing)$, i.e., without procedure calls (but with possible loops). Let Paths(S) be the set of all finite paths in S from **in** to **out**. Such a path can be formally defined as a sequence

$$p = (\mathbf{in}, \Theta_1, s_1, \Theta_2, s_2, ..., \Theta_k, \mathbf{out}),$$

where the s_i's, are its nodes and Θ_i is the instruction labelling the edge (s_{i-1}, s_i) of p.

Let I be an interpretation (with domain D) and $\mathbf{d}$ be an n-tuple of input values (in D). If there exists a successful computation of S in I with an input $\mathbf{d}$, it must follow some path in Paths(S) and yields as a result an m-tuple $\mathbf{d}'$.

The set of *pairs* $(\mathbf{d}, \mathbf{d}')$ in $D^n \times D^m$ associated in this way with some path p in Paths(S) will be denoted by p_I (note that p_I is functional).

The relation S_I defined by S in I is the union of the p_I's for p in Paths(S). Since we allow non-deterministic choices, it is not functional in general. Note also that p *is* a flowchart, with only one path from **in** to **out**.

The formal definition of p_I from p is obvious (we hope) so we omit it.

For a flowchart S over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$, we shall denote by Paths(S) the same set as before. Some of the Θ_i's occurring in paths will be procedure calls, i.e., instructions of the form call $A(\mathbf{u}; \mathbf{v})$.

We denote by $S_{I,o}$ the union of all p_I's for p in Paths$_o$(S), defined as the set of all paths in Paths(S) which do not contain any procedure call. This means that $S_{I,o}$ is the relation defined by S if one assumes that no procedure call terminates.

Of course our aim is to define S_I, the relation defined by S in I in the context of a system $\Sigma = \langle A(\mathbf{x}, \mathbf{z}) = S_A; A \in \mathscr{A} \rangle$. In Gallier (1981), Σ is considered as a graph grammar with productions $A \to S_A$, where applying a production at some edge labelled by call $A(\mathbf{u}; \mathbf{v})$ consists in substituting the "body" S_A of A for its "call," in a way which formalizes Algol's "copy rule." The notation $S \to * T$ will be used if T is obtained from S after finitely many such replacements. Then S_I is defined by

$$S_I = \bigcup \{T_{I,o}/S \to * T\}.$$

Let us make precise that the substitution of S_A for **call**$A(\mathbf{u}; \mathbf{v})$ involves

- a substitution of actual parameters $(\mathbf{u}; \mathbf{v})$ for the formal ones in S_A
- a renaming of the local variables of S_A.

See Gallier (1981) for more details. In the context of a fixed system Σ we let $A_I = (S_A)_I$.

(2.2.2) *Well-formed program schemes.* We now state the extra conditon announced in (2.1) that our flowcharts will have to satisfy. We shall require that on every computation path every variable has been assigned a value before it is used. Let S be a loop-free flowchart of rank (n, m) over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$. Let $p = (\mathbf{in}, \Theta_1, s_1, \Theta_2, ..., \Theta_k, \mathbf{out})$ be a path in S, and let v be a variable and $i \in \{0, ..., k\}$. One says that *v is defined at s_i* (with $s_0 = \mathbf{in}$, $s_k = \mathbf{out}$) iff

- either $v \in \vartheta_{\mathrm{I},n}$
- or $v \in \vartheta_{\mathrm{L}} \cup \vartheta_{\mathrm{O}}$ and v is in the left-hand side of some assignment Θ_j with $j \leq i$ or in the actual output parameter list of some procedure call Θ_j with $j \leq i$.

A variable v is *needed at s_i* iff

- either $i = k$ (i.e., $s_i = \mathbf{out}$) and $v \in \vartheta_{\mathrm{O},m}$
- or $i < k$ and Θ_{i+1} is a guard in which v occurs, or Θ_{i+1} is an assignment and v occurs in its right-hand side, or Θ_{i+1} is a procedure call and v occurs in its actual input parameter list.

A flowchart is *well-formed* if for every p in Paths(S), and every node s in p, every variable which is needed at s is defined at this node. This condition can easily be tested for each path and hence for S (since S is loop-free, it has finitely many paths). In the sequel, all flowcharts will be assumed loop-free and well-formed.

(2.3) *Operational Semantics via Attribute Grammars*

(2.3.1) *The tree of calls of a procedure evaluation.* Let us consider a loop-free (and well-formed) system

$$\Sigma = \langle A(\mathbf{x}; \mathbf{z}) = S_A / A \in \mathscr{A} \rangle.$$

Remark first that Paths(S_A) is finite for all A.

Let $P = \bigcup\{\text{Paths}(S_A)/A \in \mathscr{A}\}$ (one first ensures that Paths$(S_A) \cap$ Paths$(S_B) = \varnothing$ if $A \neq B$ by tagging with A the paths of S_A). One turns P into an $\mathscr{A}$-signature by letting $\sigma(p) = A$ if $p \in \text{Paths}(S_A)$ and $\alpha(p) = A_1 A_2 \cdots A_l$ where this word is the list of procedure names called in p (formally if $p = (\mathbf{in}, \Theta_1, s_1, ..., \Theta_k, \mathbf{out})$ then $\alpha(p) = \alpha_1 \alpha_2 \cdots \alpha_k$ with $\alpha_i = A$ if Θ_i is **call** $A(\mathbf{u}; \mathbf{v})$ for some $\mathbf{u}$, $\mathbf{v}$ and $\alpha_i = \varepsilon$ (the empty word) otherwise).

We say that a tree in $M(P)_A$ is a *tree of calls* of A. Such a tree defines the structure of recursive calls in some computation. The variables will be considered as *attributes* associated with procedure names, and appropriate *semantic rules* derived fom Σ will represent their changes of value during the computation. Furthermore, local variables will disappear; they will be symbolically evaluated into terms depending on input variables.

(2.3.2) *Construction of a conditional attribute grammar.* For A in $\mathscr{A}$ of rank (n, m) we let Inh$(A) = \{x_1, x_2, ..., x_n\}$ (its set of formal input parameters), Syn$(A) = \{z_1, z_2, ..., z_m\}$ (its set of formal output parameters), and Attr$(A) = \text{Inh}(A) \cup \text{Syn}(A)$. Let us also assume that $\{y_1, ..., y_q\}$ is the set of local variables of S_A.

We now define Φ_p for each p in Paths(S_A). This formula will be a conjunction of a set of boolean conditions and of a set of equations Γ_p following the usual restrictions concerning the definitions of attributes, so that we shall obtain a conditional attribute grammar.

Let $p = (s_0, \Theta_1, s_1, ..., \Theta_k, s_k) \in \text{Paths}(S_A)$ with $s_0 = \mathbf{in}$ and $s_k = \mathbf{out}$. Let $r_1, r_2, ..., r_l$ be the list of indices i such that Θ_i is a call. Let this call be of the form **call** $A_i(u_i; v_i)$. Hence $\alpha(p) = A_1 A_2 \cdots A_l$ and $\sigma(p) = A$. (We assume that $1 \leqslant r_1 < r_2 < \cdots < r_l \leqslant k$.)

In order to define Φ_p we need a preliminary construction. This construction will use the set $W(p)$ of attribute occurrences associated with p and Attr as defined above.

For each $i = 0, ..., k$, each variable y in $Y = \{x_1, ..., x_n, y_1, ..., y_q, z_1, ..., z_m\}$, we construct a term $t(y, i) \in M(\mathscr{F}, W(p)) \cup \{\perp\}$ representing the formal value of y at the node s_i on any computation sequence following the path p. The symbol $\perp$ is a constant standing for undefined.

Here is the definition of t:

$$t(y, i) = y(0) \qquad \text{for all} \quad y \text{ in } \{x_1, ..., x_n\}, \text{ all } i.$$

We now assume that $y \in \{y_1, ..., y_q, z_1, ..., z_m\}$:

$$t(y, 0) = \perp,$$

$$t(y, i+1) = t(y, i) \quad \text{if} \quad \Theta_{i+1} = \textbf{null} \quad \text{or } \Theta_{i+1} \text{ is a guard.}$$

Let us now assume that Θ_{i+1} is an assignment $\mathbf{u} \leftarrow (t_1, ..., t_r)$ with $\mathbf{u} = (u_1, ..., u_r)$. Then $t(y, i+1) = t(y, i)$ if $y \notin \mathbf{u}$ and $t(y, i+1) = t_j[t(y, i)/y \in Y]$ if $y = u_j$.

Let us now assume that Θ_{i+1} is **call** $B(\mathbf{u}; \mathbf{v})$ with $\mathbf{u} = (u_1, ..., u_{n'})$ and $\mathbf{v} = (v_1, ..., v_{m'})$. Then

$$t(y, i+1) = t(y, i) \quad \text{if} \quad y \notin \mathbf{v}$$

and

$$t(y, i+1) = z_j(h) \quad \text{if} \quad y = v_j \text{ and } i+1 = r_h$$

where Θ_{i+1} is the hth procedure call on the path p; note that $z_j(h)$ is an attribute occurrence.

Note that if y is defined at s_i, then $t(y, i)$ does not contain $\perp$ (more precisely $t(y, i) \in M(\mathscr{F}, W(p))$), otherwise $t(y, i) = \perp$ (because of well-formedness); this can be shown by induction on i.

We now define Φ_p as $\mathbb{AND}\ C_p$ **and** $\mathbb{AND}\ \Gamma_p$ where C_p and Γ_p are two (finite) sets of formulas defined as follows.

For every $i = 1, ..., k$ such that Θ_i is a guard, say $R(u_1, ..., u_r)$, (or **not**$(R(u_1, ..., u_t))$), one puts in C_p the condition $R(t(u_1, i), t(u_2, i), ..., t(u_r, i))$ (or the condition **not**$(R(t(u_1, i), ..., t(u_r, i)))$).

We now define Γ_p. For defining the synthesized attributes "at the root of p" (see the figures of example (2.3.4)), we put in Γ_p the semantic rules

$$z_i(0) = t(z_i, k)$$

for all $i = 1, ..., m$.

For defining the inherited attributes "at the successors of the root of p," we put in Γ_p the semantic rules

$$x_i(j) = t(u_i, r_j)$$

for all $j = 1, ..., l$ (recall that $\alpha(p) = A_1 A_2 \cdots A_l$), all $i = 1, ..., n_j$ where the rank of A_j is (n_j, m_j) and u_i is the ith actual input parameter in the corresponding procedure call, i.e., Θ_{r_j}.

Note that these semantic rules assign to formal input parameters the values of the corresponding actual input parameters. This definition makes very clear the parameter passing rule that is considered. Note also that the local variables have disappeared; they have been replaced by terms denoting their values. Examples are given below ((2.3.4), (2.4.2)).

Let G_Σ be the conditional attribute grammar associated in this way with a loop-free system Σ of recursive definitions. It is not difficult to verify that the underlying attribute grammar is of type L (See Engelfriet (1984) for a precise definition).

(2.3.3) THEOREM. *Let Σ be a loop-free system of recursive definitions. For every A in $\mathscr{A}$ of rank (n, m), for every structure $I = \langle D, (f_I)_{f \in \mathscr{F}}, (R_I)_{R \in \mathscr{R}} \rangle$, for every $\mathbf{d}$ in D^n, and every $\mathbf{d}' \in D^m$, then $(\mathbf{d}, \mathbf{d}') \in A_I$ iff there exists a tree t in $M(P)_A$ such that $(\mathbf{d}, \mathbf{d}') \in R_{t,I}$ in G_Σ.*

Sketch of the proof. Observe first that there is a one-to-one correspondence between $M(P)_A$ and $P_A = \bigcup \{\text{Paths}_0(T)/A \rightarrow {}^* T\}$ (cf. the notations of (2.2)). Roughly speaking $\langle A, P \rangle$ is the abstract context-free grammar underlying a context-free grammar generating the set P_A from start symbol A. (This is not exactly so due to the renaming and the substitutions of variables at procedure evaluation.)

Let Π_A be the corresponding mapping, associating a path in P_A with a tree in $M(P)_A$. The result follows then from:

Claim. For all t in $M(P)_A$, all $(\mathbf{d}, \mathbf{d}')$ in D^{n+m}, $(\mathbf{d}, \mathbf{d}') \in R_{t,I}$ iff $(\mathbf{d}, \mathbf{d}') \in \Pi_A(t)_I$.

This claim can be proved by structural induction on t (simultaneously for all A in $\mathscr{A}$).

(2.3.4) EXAMPLE. Consider the following sorting algorithm which modifies a sequence u (say of integers) so as to sort it (say by increasing order):

```
sort $(u; u')$:
**begin if** length$(u) > 1$ **then**
    **begin** new variable $v$, $w$, $v'$, $w'$ of type sequence of integers
    $(v; w) \leftarrow$ split $(u)$
    sort$(v; v')$
    sort$(w; w')$
    $u' \leftarrow$ merge $(v'; w')$
    **end**
**end**
```

This program uses an auxiliary procedure split (u), which divides u into two parts (as equal as possible), and produces a pair of sequences. The base function merge forms a unique sorted sequence by interleaving the two sorted sequences it takes as arguments. We can translate this program into a conditional attribute grammar with two rules shown in Fig. 5, where the effect of split is described functionally by $v = s_1(u)$ and $w = s_2(u)$.

(2.4) *Relation to Recursive Applicative Procedures*

Recursive applicative procedures have been investigated in many works (let us only quote Cadiou (1972), Greibach (1975), Vuillemin (1974, 1975), Guessarian (1981)).

The basic examples are the factorial function, the Ackermann function, and the reversal of a list. We shall not give formal definitions. The following result is well-known (see (Greibach, 1975, Theorem 7.17)).

(2.4.1) PROPOSITION. *For every recursive applicative program scheme one can construct a recursive imperative loop-free scheme equivalent for call-by-value evaluation to the given one.*

It actually extends to non-deterministic recursive applicative procedures evaluated with call-by-value as shown by Example (2.4.2) below. Through this translation, the notion of a tree of calls and the construction of G_Σ given above extend immediately to non-deterministic call-by-value recursive procedures.

(2.4.2) EXAMPLE. Here is a non-deterministic variation on the classical Ackermann function. Let f be the non-deterministic function defined as

$f(x, y) = \textbf{if } x = 0 \textbf{ then } y + 1$
  $\textbf{else if } y = 0 \textbf{ then } f(x-1, 1)$
  $\textbf{else } f(x-1, f(x, y-1)) \textbf{ or } f(x-1, y) + f(x, y-1)$

It can be translated into the following recursive imperative program $F(x, y; z)$, the effect of which being to let z take the possible values of $f(x, y)$.

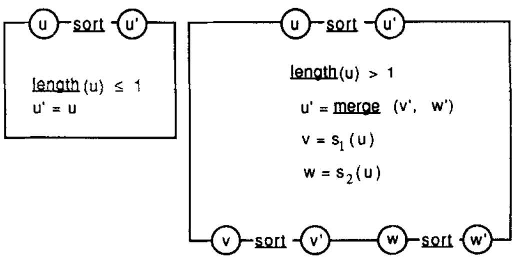

FIGURE 5

```
$F(x, y; z)$
**begin**
  **if** $x = 0$ **then** $z \leftarrow y + 1$
  **else if** $y = 0$ **then** $x' \leftarrow x - 1$; $y' \leftarrow 1$;
    **call** $F(x', y'; z')$; $z \leftarrow z'$
  **else** $\{y' \leftarrow y - 1$;
    **call** $F(x, y'; y'')$;
    $x' \leftarrow x - 1$;
    **call** $F(x', y''; z)\}$
    **or** $\{x' \leftarrow x - 1$;
      $y' \leftarrow y - 1$;
      **call** $F(x', y; u)$;
      **call** $F(x, y'; u')$;
      $z \leftarrow u + u'\}$
**end**
```

The corresponding conditional attribute grammar is shown in Fig. 6.

(2.4.3) *Call-by-name procedures.* It is well-known that for a recursive applicative program scheme Ψ and an interpretation I, the function $\Psi_{I,\text{val}}$ computed by Ψ in I under the call-by-value evaluation mechanism is in general less defined than the function $\Psi_{I,\text{name}}$ computed by the call-by-name one.

The classical example (Cadiou, 1972) is

$$\Psi(x, y) = \textbf{ if } x = 0 \textbf{ then } 0 \textbf{ else } \Psi(x - 1, \Psi(x, y)),$$

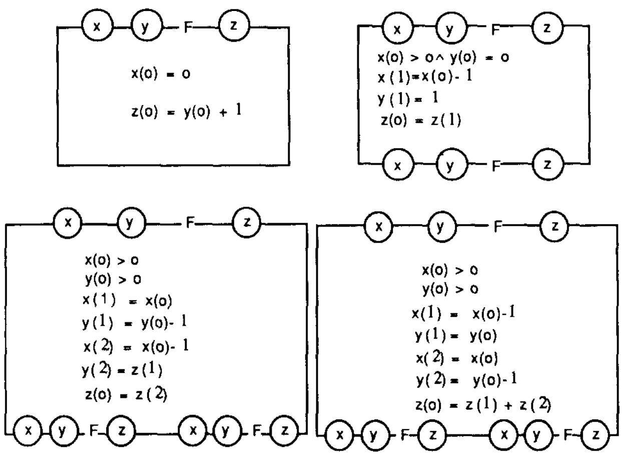

FIGURE 6

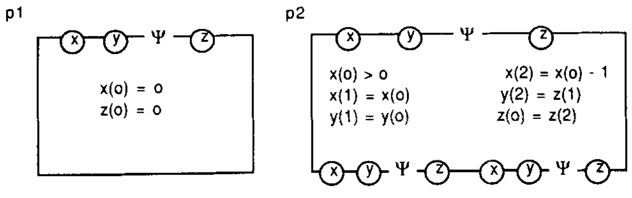

FIGURE 7

where $\Psi_{l,\text{name}}(1, 0) = 0$, $\Psi_{l,\text{val}}(1, 0)$ is undefined due to the sequence of recursive calls:

$$\Psi(1, 0) \rightarrow \Psi(0, \Psi(1, 0)) \rightarrow \Psi(0, \Psi(0, \Psi(1, 0))) \rightarrow \cdots.$$

Let us consider the relational attribute grammar associated with this recursive definition by the construction of (2.3.2). It is shown in Fig. 7. The call-by-name computation of $\Psi_l(1, 0)$ yielding 0 can be represented by the "partial" tree with undefined ($\perp$) attribute occurrences $z(1)$ and $y(2)$ of Fig. 8. This is not a tree in $M(\{p_1, p_2\})$; it corresponds to an incomplete derivation, due to the fact that $\Psi(0, \Psi(1, 0))$ evaluates to 0 without needing to evaluate the inner call (to $\Psi(1, 0)$). The classical $\perp$-trick can be used to handle this case.

(2.4.4) PROPOSITION. *Every call-by-name deterministic recursive applicative procedure can be simulated by a non-deterministic call-by-value one.*

Sketch of proof. Every domain D is augmented with a constant $\perp$ standing for "undefined." A new constant Ω is introduced to denote $\perp$. We assume for simplicity that one has a unique recursive definition:

$$\varphi(x_1, ..., x_n) = t.$$

Then one defines the new, non-deterministic definition:

$$\varphi'(x_1, ..., x_n) = t' \textbf{ or } \Omega,$$

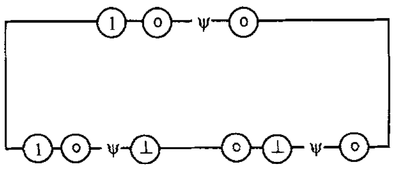

FIGURE 8

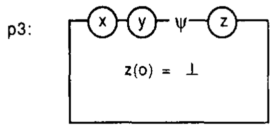

FIGURE 9

where t' is obtained from t by the substitution of φ' for φ. Then it can be shown that for $d, d_1, ..., d_n \in D$,

$$\varphi_{I,\text{name}}(d_1, ..., d_n) = d$$

if and only if

$$d \in \varphi'_{I,\text{val}}(d_1, ..., d_n).$$

(2.4.5) EXAMPLE. Again we use the example of Cadiou:

$$\Psi(x, y) = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } \Psi(x-1, \Psi(x, y)).$$

It can be simulated by

$$\Psi'(x, y) = [\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } \Psi'(x-1, \Psi'(x, y))] \textbf{ or } \Omega.$$

This recursive definition can be represented by the conditional attribute grammar consisting of p_1 and p_2 of Fig. 7 together with the production p_3 shown in Fig. 9. The call-by-name computation of Ψ can be represented by the tree of calls $p_2(p_3, p_1)$ shown with its valid assignment in Fig. 10.

(2.4.6) *Remarks.* Here is an example showing that the computation of $\Psi_{I,\text{name}}(\mathbf{d})$ by means of an attribute grammar avoids certain duplications, and hence is shorter than its evaluation by call-by-name. The reason is that the evaluation by means of attributes corresponds to a certain *sharing* of duplicated subexpressions. Consequently, this construction fails for call-by-name computations of non-deterministic schemes.

Let Σ be the equation

$$\Psi(x, y) = \textbf{if } qx \textbf{ then } f(y, y) \textbf{ else } \Psi(hx, \Psi(gx, hy)).$$

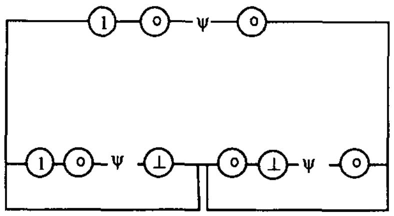

FIGURE 10

In a call-by-name computation of the form

$$\begin{aligned}\Psi(d, d') &\to \Psi(hd, \Psi(gd, hd'))\\ &\to f(\Psi(gd, hd'), \Psi(gd, hd'))\\ &\to \cdots\end{aligned}$$

the two occurrences of $\Psi(gd, hd')$ are computed separately.

In the computation by attributes one evaluates the tree shown in Fig. 11 where t' corresponds to the evaluation of $\Psi(gd, hd')$. This tree t' is *not* duplicated as was $\Psi(gd, hd')$ in the call-by-name computation. In other words, the computation by attributes corresponds to an implementation of recursivity by *sharing* as in (Vuillemin, 1975). If now the definition of $\Psi(x, y)$ is augmented by some alternative cases making it non-deterministic, the two occurrences of $\Psi(gd, hd')$ may be evaluated into two distinct values and this cannot be handled by means of the attribute grammar.

(2.5) *Relation to Logic Programming*

We recall from Deransart and Maluszynski (1985) some results on the translation of a definite clause program into a relational attribute grammar. The semantics of logic programs can be formalized by means of *proof trees* which in turn can be viewed as trees decorated by attributes. We illustrate the definitions and constructions using a logical version of Example (2.3.4).

(2.5.1) DEFINITIONS. The idea of logic programming concerns computing relations specified by logic formulas. The formulas are restricted to

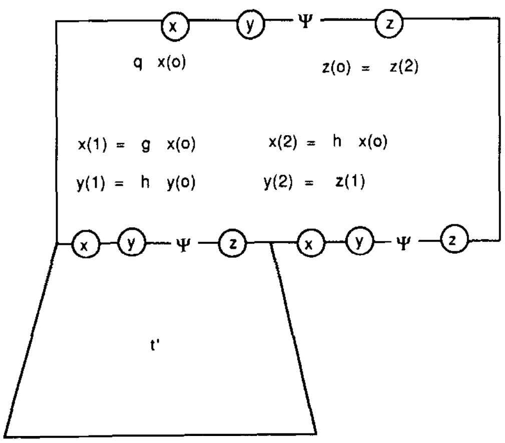

FIGURE 11

definite clauses. A definite clause is a pair consisting of an atomic formula A and a finite set of atomic formulas $\{B_1, ..., B_q\}$, $q \geq 0$, written as

$$A \Leftarrow B_1, ..., B_q \qquad \text{(and also } A \Leftarrow \text{ in case } q = 0\text{).}$$

The atomic formulae are constructed, as usual, from predicate letters and (one-sorted) terms: A is an atomic formula iff it is of the form $p(t_1, ..., t_n)$ where p is an n-ary predicate and $t_1, ..., t_n$ are terms.

A definite clause of the form described above can be represented in the standard logic notation as the formula

$$\forall x_1 \cdots \forall x_k (B_1 \wedge \cdots \wedge B_q \Rightarrow A),$$

where $x_1 \cdots x_k$ are all variables occurring in the clause.

A *definite clause program* (DCP) is a triple $\mathscr{C} = \langle \mathscr{N}, \mathscr{F}, \mathscr{P} \rangle$ where $\mathscr{N}$ is a finite set of predicates letters with assigned ranks, $\mathscr{F}$ is a set of function symbols with assigned ranks, $\mathscr{P}$ is a finite set of definite clauses constructed with $\mathscr{N}$ and $\mathscr{F}$, and ϑ is a denumerable set of variables.

EXAMPLE. The logical version of the sorting program (2.3.4) is the following (we use the same names prefixed by p- recalling that the names of the functions are translated into predicate names). Lists are dotted as usual:

$$\mathscr{N} = \{p\text{-sort}, p\text{-split}, p\text{-merge}\}$$

$$\mathscr{F} = \{\text{nil}, \cdot\}$$

$$\mathscr{P} =$$

1. p-sort (nil, nil) $\Leftarrow$
2. p-sort ($A \cdot$ nil, $A \cdot$ nil) $\Leftarrow$
3. p-sort ($A \cdot B \cdot L$, S) $\Leftarrow$ p-split ($A \cdot B \cdot L$, $L1$, $L2$),
   p-sort ($L1$, $S2$),
   p-sort ($L2$, $S2$),
   p-merge ($S1$, $S2$, S)

p-split and p-merge are not described here; variables begin with upper-case letters.

The meaning of each predicate is the same as in (2.3.4) provided that p-$f(a_1, ..., a_n)$ is equivalent to $a_n = f(a_1, ..., a_{n-1})$.

Usually a DCP is considered as a specification of its least Herbrand model (see, e.g, (Apt, 1982)). It was shown in (Clark, 1979; Ferrand, 1987) that one can deal instead with the set of all atomic formulas (not necessarily the ground ones) which are logical consequences of the clauses of the DCP. This set is called its *denotation* in (Ferrand, 1987). Each element of this set can be obtained by constructing a *proof tree.* For the purposes of this paper it is convenient to consider a DCP to be the specification of the set of all proof trees (thus the denotation is the set of all the proof tree roots).

We introduce now some auxiliary notions and the notion of proof tree. A *substitution* is an operation on expressions (terms or formulas), which replaces all occurrences of a variable in an expression by a term. The result is called an *instance* of the expression. A substitution can be seen as a term-assignment.

A *proof tree* is an ordered labelled tree whose labels are atomic formulas (possibly including variables) or are empty. The set of proof trees of a given DCP $\mathscr{C}$ is defined as follows:

1. If $A \Leftarrow$ is an instance of a clause of $\mathscr{C}$ then the tree consisting of two vertices whose root is labelled A and whose only leaf has the empty label (ε) is a proof tree.

2. If $T_1, ..., T_q$ for some $q > 0$ are proof trees with roots labeled $B_1, ..., B_q$ and if $A \Leftarrow B_1, ..., B_q$ is an instance of a clause of $\mathscr{C}$, then the tree consisting of the root labeled with A and the subtrees $T_1, ..., T_q$ is a proof tree.

By a *partial proof tree* we mean any finite tree constructed by "pasting together" instances of clauses. Thus a proof tree is a partial proof tree whose leaves all have empty labels.

An example of partial proof tree is shown in Fig. 12.

(2.5.2) *Transformation of DCPs into relational attribute grammars.* We outline from Deransart and Maluszynski (1985) the construction, without formal description, illustrating it by some examples. The idea is the following:

– The set of clauses can be viewed as an abstract context-free grammar, if we forget all term arguments of the predicates.

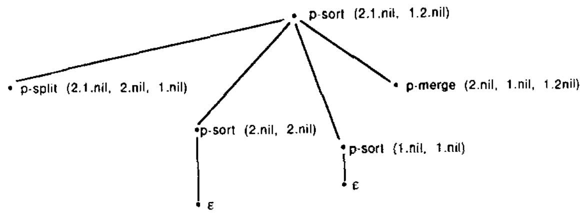
p-sort (2.1.nil, 1.2.nil)
p-split (2.1.nil, 2.nil, 1.nil)
p-merge (2.nil, 1.nil, 1.2nil)
p-sort (2.nil, 2.nil)
p-sort (1.nil, 1.nil)
ε
ε

FIGURE 12

– Each argument can be viewed as an attribute (denoted *pi* for the *i*th argument of predicate *p*).

– Each relation Φ_p associated to every grammatical rule can be defined using a canonical interpretation defined as follows: to every clause *c* (and thus every grammatical rule) is associated a relational symbol r_c of rank *n* which is equal to the number of all attribute occurrences in the rule (all argument occurrences in the clause), whose interpretation is the set of all instances of the atom $r_c\ (a_1, ..., a_n)$ where a_i, $1 \leq i \leq n$, is the term appearing in the corresponding place in the clause.

EXAMPLE. Relational attribute grammar corresponding to the sorting program (see Fig. 13):

$N = \{p\text{-sort}, p\text{-split}, p\text{-merge}\}$

$P = \{p1, p2\text{: } \langle \varepsilon, p\text{-sort}\rangle, p3\text{: } \langle p\text{-split } p\text{-sort}$

$p\text{-sort } p\text{-merge}, p\text{-sort}\rangle\}$

Attr(p-sort) = {p-sort1, p-sort2}

Attr(p-split) = {p-split1, p-split2, p-split3} etc.

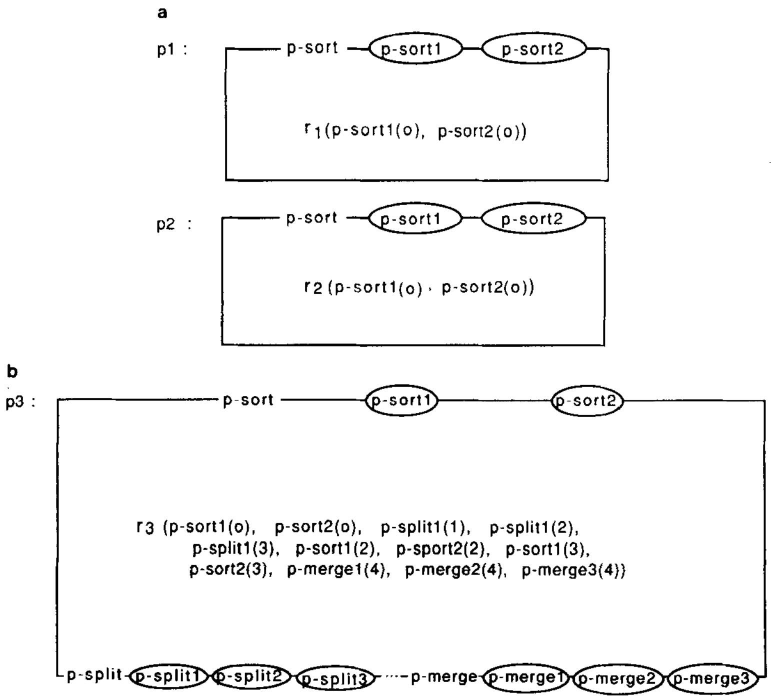
a
p1 :
p-sort
p-sort1
p-sort2
r1(p-sort1(o), p-sort2(o))
p2 :
p-sort
p-sort1
p-sort2
r2 (p-sort1(o) · p-sort2(o))
b
p3 :
p-sort
p-sort1
p-sort2
r3 (p-sort1(o), p-sort2(o), p-split1(1), p-split1(2), p-split1(3), p-sort1(2), p-sport2(2), p-sort1(3), p-sort2(3), p-merge1(4), p-merge2(4), p-merge3(4))
p-split
p-split1
p-split2
p-split3
p-merge
p-merge1
p-merge2
p-merge3

FIGURE 13

$\mathbb{D}$ is $\{r1(\text{nil}, \text{nil}),$

$r2(A \cdot \text{nil}, A \cdot \text{nil}),$

$r3(A \cdot B \cdot L, S, A \cdot B \cdot L, L1, L2, L1, S1, L2, S2, S1, S2)$

for any $A, B, L, S, L1, L2, S1, S2$ replaced by terms of $M(\mathcal{F}, \mathcal{V})\}$.

In order to simplify this notation, it will be shortened by avoiding to denote explicitly the relational symbols and by replacing the attribute names, which are obvious if the attribute places are ordered, by the corresponding term in the clause. The relational attribute grammar is shown on Fig. 14.

(2.5.3) *Main result on the transformation.* The following theorem is proved in Deransart and Maluszynski, (1985):

THEOREM. *Let* $\mathscr{C} = \langle \mathcal{N}, \mathcal{F}, \mathcal{P} \rangle$ *be a DCP and let* $G_{\mathscr{C}} = \langle N, P, \text{Attr}, \Phi, \mathbb{D} \rangle$ *be the relational attribute grammar obtained by the outlined construction. Then the set of proof trees of* $\mathscr{C}$ *is isomorphic to the set of trees with valid assignments of* $G_{\mathscr{C}}$.

This theorem states that results obtained in one representation can be transposed into the other representation. Thus all the results obtained using the representation by a relational attribute grammar yield results on the proof trees.

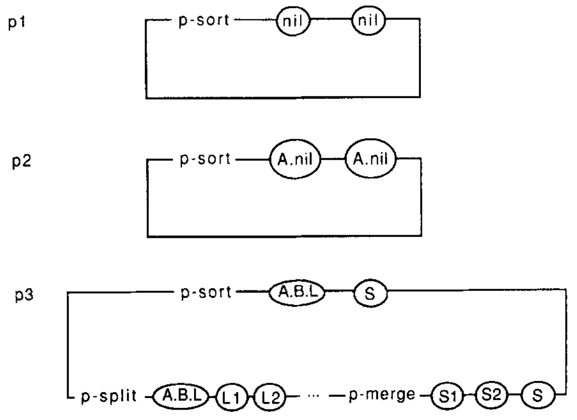

FIGURE 14

(3) Fix-Point Induction for Attribute Grammars

We introduce specifications for attribute grammars, and the *correctness* of an attribute grammar w.r.t a specification which is akin to the partial correctness of programs.

We introduce a proof method making possible to establish the correctness of an attribute grammar w.r.t. a specification. This method is sound. It is complete in an abstract sense, i.e., if one uses the "maximum" logical language where every relation on the domain of interpretation can be represented by a formula. This completeness result is analogous to the theorem of De Bakker and Meertens (1975) stating the completeness of the inductive assertion method for flowcharts. But the incompleteness result of Wand (1978) extends to our proof method, when one restricts formulas to first-order logic.

(3.1) *Specifications*

(3.1.1) Definition. Given a relational attribute grammar $G = \langle N, P, \mathrm{Attr}, \Phi, D\rangle$, a specification for G consists in a family $\Theta = \{\Theta^X\}_{X \in N}$ of formulas belonging to some logical language $\mathscr{L}'$ including $\mathscr{L}$ ($\mathscr{L}$ is the language where Φ is defined).

Each formula has its free variables in the set $\mathrm{Attr}(X)$. We shall also write $\Theta^X(a_1, ..., a_m)$ to recall this, where $\{a_1, ..., a_m\} = \mathrm{Attr}(X)$.

We say that G *is correct w.r.t. the specification* Θ or *satisfies* Θ, or that Θ *is valid for* G if, for every tree t in $M(P)$ and for every valid t-assignment v,

$$\mathbb{D} \models \Theta^X(v(a_1(0), ..., v(a_m(0))) \quad (\text{i.e., } (\mathbb{D}, v') \models \Theta^X \text{ where } v'(a_i) = v'(a_i(0))),$$

where $X = \sigma(t)$.

In a practical situation, Θ^S is given where S is some initial non-terminal and G must be found to satisfy $(\Theta^S, \mathbf{true}, ..., \mathbf{true})$. But for the purpose of a theoretical investigation we shall prefer to start from G and investigate the set of specifications which G satisfies. For this reason we shall use the upside-down terminology "Θ is valid for G."

The following proposition says that the validity of Θ^X holds not only at the root of a tree, but also at all nodes, if Θ is valid.

(3.1.2) Proposition. *If Θ is valid for G then, for all tree t in $M(P)$, for all valid t-assignment v,*

$$\mathbb{D} \models \Theta^X(v(a_1(u)), ..., v(a_m(u)))$$

for every u in $\mathrm{Node}(t)$ *where $X = \sigma_t(u)$ and* $\{a_1, ..., a_m\} = \mathrm{Attr}(X)$.

Proof. Let t, v, u be given as in the statement. Let $t' = t/u$; then the assignment v': $W(t') \to D$ such that $v'(a(u')) = v(a(uu'))$ for all $a(u')$ in $W(t')$ is a valid t'-assignment. Whence the result by Definition (3.1.1).

(3.1.3) *Remark.* Saying that G satisfies Θ corresponds to the *partial* $\forall$-*correctness* of a program w.r.t. a specification, as defined by Gallier (1981).

Other notions, in particular the total correctness, are of obvious interest. In particular, one could define the *total* $\exists$-*correctness of* G *w.r.t.* Θ by requiring the existence of at least one tree t in $M(P)_X$, at least one valid t-assignment v such that $v(b_i(0)) = d_i$ for every l-tuple $(d_1, ..., d_l)$ satisfying some "input condition" $\phi^X(d_1, ..., d_l)$, (where $\{b_1, ..., b_l\}$ is a subset of Attr(X)) and satisfying Θ. See (Deransart, 1984) for the investigation of such a concept.

(3.1.4) Definition. Let Θ and Θ' be two specifications for the same relational attribute grammar G. One says that Θ *is weaker than* Θ' (Θ' *is stronger than* Θ), denoted by $\Theta' \Rightarrow \Theta$, if for all X,

$$\mathbb{D} \models (\Theta'^X \Rightarrow \Theta^X).$$

Note that Θ and Θ' can be written in different logical languages. If Θ' is valid for G and $\Theta' \Rightarrow \Theta$ then Θ is valid for G.

The specification TRUE (every formula of which is identical to **true**) is always valid for G in a trivial way. It is the weakest valid specification for G.

There exists a strongest valid specification for G, belonging to the "language" $\mathscr{L}_{\mathbb{D}}$ of all relations on D. Let us denote it by Θ_G and define it as follows (with Attr(X) $= (a_1, ..., a_m)$): $(\mathbb{D}, d_1, ..., d_m) \models \Theta_G^X(a_1, ..., a_m)$ if and only if there exists t in $M(P)_X$, a valid t-assignment v such that $v(a_i(0)) = d_i$ for all $i = 1, ..., m$. (In the first formula d_i is assigned to a_i for all $i = 1, ..., m$.)

In a more intuitive way, Θ_G^X is defined as $\bigcup \{R_{t,\mathbb{D}}/t \in M(P)_X\}$. It should be noted that Θ_G^X is not necessarily expressible in first-order logic (see (3.3) below). From the definition, it is clear that a specification Θ is valid for G iff it is weaker than Θ_G (i.e., $\Theta_G \geqslant \Theta$).

(3.2) *How to Establish the Correctness of an Attribute Grammar with Respect to a Specification*

(3.2.1) Definition. Let G be a relational attribute grammar $\langle N, P$, Attr, Φ, $\mathbb{D}\rangle$ and Θ be a specification for G. One says that Θ is *inductive* if, for all p in P,

$$\mathbb{D} \models \Phi_p \text{ and } \Theta_1^{X_1} \text{ and } \Theta_2^{X_2} \cdots \text{ and } \Theta_n^{X_n} \Rightarrow \Theta_0^{X_0},$$

where $\langle X_1 \cdots X_n, X_0 \rangle$ is the profile of p and for every formula ϕ in $\mathscr{L}(\mathrm{Attr})$, ϕ_i denotes the formula $\phi[a(i) | a \in \mathrm{Attr}]$ that belongs to $\mathscr{L}(\{a(i) | a \in \mathrm{Attr}, i \in \mathbb{N}\})$.

(3.2.2) PROPOSITION. *If Θ is inductive then Θ is valid for G.*

Proof. One can prove by structural induction the validity of the proposition,

$$\forall X \in N, \forall t \in M(P)_X, \mathbb{D} \models \Phi_t \Rightarrow \Theta^X.$$

(3.2.3) *Remark.* A specification may be valid without being inductive as shown by the following example:

$$N = \{X, Y\}$$

$$P = \{p, q\} \qquad \text{with} \quad p\colon \langle Y, X \rangle \text{ and } q\colon \langle \varepsilon, Y \rangle$$

$$\mathrm{Attr}(X) = \{a\}$$

$$\mathrm{Attr}(Y) = \{b\}$$

$$\Phi_p(a(0), b(1)) \Leftrightarrow a(0) = b(1)$$

$$\Phi_q(b(0)) \Leftrightarrow b(0) = 0$$

$$\mathbb{D} = \mathbb{N}.$$

Let Θ be the specification such that

$$\Theta^X(a) \Leftrightarrow a = 0$$

$$\Theta^Y(b) \Leftrightarrow \mathbf{true}.$$

It is clearly valid but not inductive since the following does not hold in $\mathbb{N}$:

$$\forall a(0), b(1) \; [a(0) = b(1) \textbf{ and true} \Rightarrow a(0) = 0].$$

(3.2.4) PROPOSITION. *The strongest valid specification for G, namely Θ_G, is inductive.*

Proof. Let $P \in P_{\langle X_1 \cdots X_n, X_0 \rangle}$. Let $\Theta = \Theta_G$. Let v be any assignment: $W(p) \to D$. We must verify that

$$(\mathbb{D}, v) \models \Phi_p \textbf{ and } \Theta_1^{X_1} \textbf{ and} \cdots \textbf{and } \Theta_n^{X_n} \Rightarrow \Theta_0^{X_0}.$$

Let us assume that the left part of the implication is true. By definition of Θ ($= \Theta_G$) there is a tree t_i for each $i = 1, ..., n$ and a valid t_i-assignment v_i such that $v_i(a(0)) = v(a(i))$ for all a in $\mathrm{Attr}(X_i)$. Let $t = p(t_1, ..., t_n)$ and v' be the t-assignment such that $v'(a(0)) = v(a(0))$ for a in $\mathrm{Attr}(X_0)$ and $v'(a(iu)) = v_i(a(u))$ for all $a(u)$ in $W(t_i)$. Since v satisfies Φ_p, v' satisfies Φ_t and hence v' (and v) satisfy $\Theta_0^{X_0}$ by the definition of Θ_G.

(3.2.5) THEOREM. *Let G be a relational attribute grammar. A specification Θ for G is valid iff it is weaker than some inductive specification Θ'.*

Proof and remarks. The "if" part follows from Proposition (3.2.2). It corresponds to the *soundness* of the proof method consisting in the following steps:

(1) defining a specification Θ'

(2) proving that Θ' is inductive

(3) proving that $\Theta' \Rightarrow \Theta$.

The "only if" part follows from Proposition (3.2.4). It corresponds to a *completeness theorem w.r.t. the language of all relations on* $\mathbb{D}$. One does not have the completeness if one restricts assertions to some logical language like first-order logic. See (3.3) below.

(3.2.6) EXAMPLES. Coming back to Examples (1.5.1) and (1.4.3), it is not difficult to prove the correctness of Katayama–Hoshino's example with respect to the inductive specification such that:

$$\Theta^S \Leftrightarrow \exists n, k = 4 * n$$

$$\Theta^A \Leftrightarrow \exists n\,(g = f + 2 * n \wedge k = h + 2 * n).$$

In the same way it is not difficult to prove the equivalence of the two attribute definitions given in Example (1.4.3), using the following inductive specification (note that both have the same underlying abstract context-free grammar),

$$\Theta^S \Leftrightarrow (\mathbf{baword}(x) \text{ or } \mathbf{bword}(x)) \text{ and } x = y$$

$$\Theta^A \Leftrightarrow (\mathbf{bword}(y) \text{ or } \mathbf{baword}(y))$$
$$\mathbf{and}(\mathbf{bword}(u) \Rightarrow x = uy) \text{ and } (\mathbf{baword}(u) \Rightarrow x = u).$$

Note that this specification may appear relatively complicated and the proof even if not difficult needs to be well-organized in order to remain sufficiently clear. The method presented in (4) can be viewed as an aid to make clearer demonstrations.

(3.3) *Wand's Incompleteness Result*

We show an example of a relational attribute grammar and a valid specification Θ for which no inductive specification Θ' written in first-order logic can be found to prove it (i.e., such that $\Theta' \Rightarrow \Theta$).

Our example is a straightforward translation of an example of Wand (1978) showing the incompleteness of Hoare's logic when invariant assertions are written in first-order logic.

Let p, q, r be unary predicates, f be a unary function symbol, and x, y be variables. Let $\mathbb{D}$ be the first-order structure such that its domain D is $\{a_n/n \geq 0\} \cup \{b_n/n \geq 0\}$, and:

$$
\begin{array}{lll}
f(a_0) = a_0, & f(b_0) = b_0, & \\
f(a_n) = a_{n-1}, & f(b_n) = b_{n-1} & \text{for} \quad n \geq 1, \\
p(x) = \textbf{true} & \text{iff} \quad x = a_0 & \\
q(x) = \textbf{true} & \text{iff} \quad x = b_0 & \\
r(x) = \textbf{true} & \text{iff} \quad x = a_n \text{ for some } n \text{ of the form } k(k+1)/2. &
\end{array}
$$

It is shown in (Wand, 1978, Theorem 2) that there is no first-order formula ϕ with one free variable x in such that, for d in D, $(\mathbb{D}, d) \models \phi$ iff $d \in \{a_n/n \geq 0\}$.

Let G be the following relational attribute grammar:

$N = \{A, X\}$

$P_{\langle X,A \rangle} = \{a\}, \qquad P_{\langle X,X \rangle} = \{b\}, \qquad P_{\langle \varepsilon,X \rangle} = \{c\}$

$\text{Attr}(A) = \text{Attr}(X) = \{x, y\}$

Φ_a is the formula $\quad r(x(0))$ **and** $x(0) = x(1)$ **and** $y(0) = y(1)$

Φ_b is the formula $\quad$ **not** $(p(x(0)))$ **and** $($**not**$(q(x(0))))$
and $x(1) = f(x(0))$ **and** $y(0) = y(1)$

Φ_c is the formula $\quad (p(x(0))$ **or** $q(x(0)))$ **and** $x(0) = y(0)$.

Let $\mathbb{D}$ be the above-defined interpretation.

It is easy to see that the strongest specification Θ_G is the following:

$$
\begin{array}{lll}
\Theta^X_G(x, y) & \text{iff} & (x \in \{a_n/n \geq 0\} \text{ and } y = a_0) \\
 & \text{or} & (x \in \{b_n/n \geq 0\} \text{ and } y = b_0) \\
\Theta^A_G(x, y) & \text{iff} & (x \in \{a_n/n = k(k+1)/2 \text{ for some } k \geq 0\} \\
 & \text{and} & y = a_0).
\end{array}
$$

Let Θ be the valid specification such that:

$$\Theta^X(x, y) \text{ is } \textbf{true}$$

$$\Theta^A(x, y) \text{ is } p(y).$$

Assume there is an inductive specification Θ' such that $\Theta' \Rightarrow \Theta$. It must satisfy the following conditions (Definition (3.2.1)):

(1) $\Theta'^A(x, y) \Rightarrow p(y)$ (i.e., $\Theta' \Rightarrow \Theta$)

(2) $\Theta'^X(x, y)$ **and** $r(x) \Rightarrow \Theta'^A(x, y)$

(3) $\Theta'^X(f(x), y)$ **and** (**not** $(p(x))$) **and** (**not** $(q(x))) \Rightarrow \Theta'^X(x, y)$

(4) $(p(x)$ **or** $q(x))$ **and** $x = y \Rightarrow \Theta'^X(x, y)$.

Let now ϕ be the formula $\forall y[\Theta'^A(x, y) \Rightarrow p(y)]$.

Claim. $(\mathbb{D}, d) \models \phi$ iff $d \in \{a_n / n \geq 0\}$.

We first note some facts. From (4) the following holds:

(5) $\Theta'^X(b_0, b_0)$.

From (5) and (3) the following holds:

(6) $\Theta'^X(b_n, b_0)$ for all $n \geq 0$.

If $\Theta'^X(a_n, d')$ holds then $\Theta'^X(a_m, d')$ holds for all $m \geq n$ (by (3)). Let us choose $m \geq n$ of the form $k(k+1)/2$; then $\Theta'^A(a_m, d')$ holds by (2); hence $p(d')$ holds by (1). This shows that $(\mathbb{D}, d) \models \phi$ for all d in $\{a_n / n \geq 0\}$.

Now let $d = b_n$ for any n. Then $\Theta'^X(b_n, b_0)$ holds by (6); since $p(b_0)$ does not hold, $(\mathbb{D}, d) \models \phi$ does not hold.

The claim is proved. But ϕ (hence Θ') cannot be first-order formulas.

(3.4) *Application to the Partial Correctness of Imperative Schemes*

Let (Σ, A) be a scheme over $(\mathscr{F}, \mathscr{R}, \mathscr{A})$ of rank (n, m). Let $\phi(\mathbf{x})$ and $\psi(\mathbf{x}, \mathbf{z})$ be formulas of some logical language $\mathscr{L}$ constructed over $\mathscr{F}$ and $\mathscr{R}$.

Let I be an interpretation with domain D. We say that A is *partially correct* w.r.t. (ϕ, ψ) in I for all $\mathbf{d}$ in D^n, all $\mathbf{d}'$ in D^m, if $\phi(\mathbf{d})$ holds, if $(\mathbf{d}, \mathbf{d}') \in A_I$, then $\psi(\mathbf{d}, \mathbf{d}')$ holds. $\phi(\mathbf{d})$ holds means $(I, \mathbf{d}) \models \phi(x)$ and similarly $\psi(\mathbf{d}, \mathbf{d}')$ holds means $(I, \mathbf{d}, \mathbf{d}') \models \psi(y, z)$. This property is called $\forall$-*partial correctness* in (Gallier, 1981) where other correctness conditions were considered.

Given (Σ, A) and (ϕ, ψ), we define a specification Θ for G_Σ, the conditional attribute grammar defined in (2.3.2), by letting:

$$\Theta^A: \phi(\mathbf{x}) \Rightarrow \psi(\mathbf{x}, \mathbf{z})$$

$$\Theta^B: \textbf{true}$$

for all B in $\mathscr{A}$, $B \neq A$.

For any interpretation I, it is clear from Theorem (2.3.3) that (Σ, A) is partially correct w.r.t. (ϕ, ψ) iff Θ is valid for G_Σ.

Hence the proof method derived from Theorem (3.2.5) can be applied to G_Σ and Θ and yields a sound and complete method for establishing the partial correctness of (Σ, A) w.r.t. (ϕ, ψ), which is nothing else than the classical fix-point induction extended in (Gallier, 1981) in order to deal with non-loop-free systems.

(3.4.1) EXAMPLE. Let $A(x; z) = S_A$ be a recursive definition such that S_A is the flowchart shown in Fig. 15. The interpretation is the domain of integers with the usual operations.

We shall prove the partial correctness of A with respect to $(\phi(x), \psi(x, z))$ where $\phi(x)$ is "$x \geq 0$", and $\psi(x, z)$ is "$z \geq 1$ **and** $\lceil (x+1)/2 \rceil$ divides z" (where $\lceil y \rceil$ denotes the least integer y' such that $y' \geq y$).

Since $A(x; z)$ defines $z = x!$ whenever $x \geq 0$, its partial correctness follows from an easy arithmetical argument ($\lceil (x+1)/2 \rceil \leq x$). Gallier's method consists in finding predicates $\phi_A(x)$, $\psi_A(x, z)$, and $q_\alpha(x, y_1)$ such that, for all x, y_1, y_2,

(1) $\phi(x) \Rightarrow \phi_A(x)$

(2) $\phi_A(x)$ **and** $x = 0 \Rightarrow \psi_A(x, 1)$

(3) $\phi_A(x)$ **and** $x > 0 \Rightarrow q_\alpha(x, x-1)$ **and** $\phi_A(x-1)$

(4) $q_\alpha(x, y_1)$ **and** $\psi_A(y_1, y_2) \Rightarrow \psi_A(x, y_2 \cdot x)$

(5) $\phi(x)$ **and** $\psi_A(x, z) \Rightarrow \psi(x, z)$.

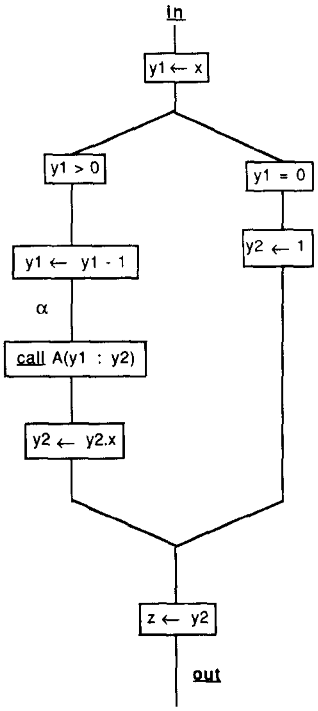

FIGURE 15

Intuitively ϕ_A is an input predicate and ψ_A is an input–output predicate for A such that A is partially correct w.r.t. ϕ_A, ψ_A. The predicate $q_\alpha(x, y_1)$ is a relation that is satisfied for every computation fom **in** to α in S_A (by (3)) and is sufficient to ensure the validity (with respect to ψ_A) of every computation in S_A from α to **out** (by (4)).

Conditions (1) and (5) say that (ϕ_A, ψ_A) is stronger than (ϕ, ψ). Condition (2) states the correctness w.r.t. (ϕ_A, ψ_A) of the computations of S_A following the right-most path (see Fig. 15). Conditions (3) and (4) state the same thing for the computations following the left-most path, if one assumes the correctness of the call to A on this path, and by using q as an intermediate predicate.

Note that q_α can be eliminated between (3) and (4), i.e., that the conjunction of (3′) and (4′) is sufficient to establish the partial correctness of A w.r.t. (ϕ, ψ):

(3′) $\phi_A(x)$ **and** $x > 0 \Rightarrow \phi_A(x-1)$

(4′) $\phi_A(x)$ **and** $x > 0$ **and** $\psi_A(x-1, y_2) \Rightarrow (x, y_2 \cdot x)$.

The completeness theorem of Gallier (1981) shows that one can take for ϕ_A the predicate **true** and for ψ_A the input-output relation defined by A, i.e., $\psi_A(x, z)$ iff $x \geq 0$ **and** $z = x!$.

The following formula being false (in $\mathbb{Z}$),

$$\forall x, y[x, y > 0 \textbf{ and } \lceil x/2 \rceil \text{ divides } y \Rightarrow \lceil (x+1)/2 \rceil \text{ divides } x \cdot y] \qquad (*)$$

(it suffices to take $x = 12$, $y = 6$), one cannot simply take ψ instead of ψ_A. This is an example of the frequent situation where for doing a proof by induction, one needs an inductive assertion which is stronger than the one to be established.

Let us now do this proof of partial correctness in terms of attribute grammars. The relational attribute grammar associated with A consists of two "productions." The first is p_1 with profile $\langle A, A \rangle$. The associated relation Φ_{p_1} is

$$x(0) > 0 \textbf{ and } x(1) = x(0) - 1 \textbf{ and } z(0) = z(1) \cdot x(0).$$

The second is p_2 of profile $\langle \varepsilon, A \rangle$ with relation Φ_{p_2},

$$x(0) = 0 \qquad \textbf{and} \qquad z(0) = 1.$$

The specification (ϕ, ψ) is now written as a unique formula $\phi \Rightarrow \psi$ that we denote by Θ and is

$$x \geq 0 \Rightarrow z \geq 1 \qquad \textbf{and} \qquad \lceil (x+1)/2 \rceil \text{ divides } z.$$

Since (∗) is false in $\mathbb{Z}$ so is

$$\Phi_{p_1} \textbf{ and } \Theta[x(1)/x, z(1)/z] \Rightarrow \Theta[x(0)/x, z(0)/z].$$

Hence Θ is not inductive.

The completeness theorem (3.2.5) shows that in order to prove the validity of the relational attribute w.r.t. Θ (hence the partial correctness of A w.r.t. (ϕ, ψ)), it suffices to prove $\Theta_0 \Rightarrow \Theta$ were Θ_0 is the strongest valid specification, which is nothing else by Theorem (2.3.3) than the input–output relation of A.

It follows that Θ_0 coincides with $\phi_A \Rightarrow \psi_A$ where ϕ_A, ψ_A follow from Gallier's completeness proof.

The inductivity of Θ_0 is equivalent to the conjunction of (2), (3′), (4′).

(3.5) *Application to Logic Programming*

As shown by Theorem (2.5.4) the inductive and complete proof method depicted in this section can be applied to the correctness of definite clause programs.

A *specification* is given by a *family of formulas* (expressed in some logical language which can be a higher level than the program's logic or informally expressed properties if a formal proof is not required), one formula *associated with each predicate name*. It is *valid* if it holds at all proof tree roots (all terms in the atomic formulas of the denotation make the specification valid, i.e., all possible answers satisfy the specification), thus a DCP will be said to be partially correct w.r.t. the (valid) specification.

The associated attribute grammar obtained by the construction (2.5.2) uses the canonical termal interpretation. Now we are interested in interpreting the axioms in some non termal structure $\mathbb{D}$ of the specification's language. The proof method for attribute grammars can thus be applied directly by showing that some specification is inductive, that is in every rule c the formula (3.2.1) holds, in which the free variables of $\Theta_i^{X_i}(i \geqslant 0)$ are substituted by the corresponding terms in the clause c and the remaining variables are all universally quantified. It is easy to see that every logical consequence in $\mathbb{D}$ of an inductive specification is valid in the sense defined above.

For example, consider the following program (list permutations):

(1) $\text{perm}(\text{nil}, \text{nil}) \Leftarrow$

(2) $\text{perm}(A \cdot L, B \cdot M) \Leftarrow \text{extract}(A \cdot L, B, N), \text{perm}(N, M)$

(3) $\text{extract}(A \cdot L, A, L) \Leftarrow$

(4) $\text{extract}(A \cdot L, B, A \cdot M) \Leftarrow \text{extract}(L, B, M)$

It can be proved partially correct with regard to the specification

$$\Theta^{\text{perm}}(p_1, p_2)\text{: list } p_1 \Rightarrow (\text{list } p_2 \wedge p_2 \text{ is a permutation of } p_1),$$

where p_1 and p_2 denote respectively the first and the second arguments of the relation "perm," and

$$\Theta^{\text{extract}}(e_1, e_2, e_3)\text{: list } e_1 \Rightarrow \exists L_1, L_3(e_1 = L_1 \cdot (e_2 \cdot L_3) \wedge e_3 = L_1 \cdot L_3),$$

where $\cdot$ denotes list concatenation. The specification $\{\Theta^{\text{perm}}, \Theta^{\text{extract}}\}$ is inductive.

The formulas to be proved are:

$$
\begin{array}{ll}
(1) & \text{list(nil)} \Rightarrow (\text{list(nil)} \wedge \text{nil is a permutation of nil}) \\
(2) & \text{list}(N) \Rightarrow (\text{list}(M) \wedge M \text{ is a permutation of } N) \\
\wedge & \text{list}(A \cdot L) \Rightarrow \exists L_1, L_3(A \cdot L = L_1 \cdot (B \cdot \text{nil}) \cdot L_3) \wedge N = L_1 \cdot L_3 \\
\Rightarrow & [\text{list}(A \cdot L) \Rightarrow (\text{list}(B \cdot M) \wedge B \cdot M \text{ is a permutation of } A \cdot L)] \\
(3) & \text{list}(A \cdot L) \Rightarrow (\exists L_1, L_3(A \cdot L = L_1 \cdot (A \cdot \text{nil}) \cdot L_3) \wedge L = L_1 \cdot L_3) \\
(4) & [\text{list}(L) \Rightarrow (\exists L_1, L_3(L = L1 \cdot (B \cdot \text{nil}) \cdot L_3) \wedge M = L_1 \cdot L_3)] \\
\Rightarrow & [\text{list}(A \cdot L) \Rightarrow (\exists L'_1, L'_3(A \cdot L = L'_1 \cdot (B \cdot \text{nil}) \cdot L'_3) \wedge A \cdot M = L'_1 \cdot L'_3)]
\end{array}
$$

Now we consider the usual Peano's axioms for the addition on the natural integers $\mathbb{N}$ defined with

$$
\begin{aligned}
\mathscr{F} &= \{o, s\}, \\
\mathscr{P} &= \text{plus}(o, x, x) \Leftarrow \\
&\quad \text{plus}(sx, y, sz) \Leftarrow \text{plus}(x, y, z)
\end{aligned}
$$

Its translation into a relational attribute grammar is

$$
\begin{aligned}
N &= \{\text{plus}\} \\
P &= \{p1\text{: } \langle \varepsilon, \text{plus} \rangle, p2\text{: } \langle \text{plus}, \text{plus} \rangle\} \\
\text{Attr(plus)} &= \{\text{plus1}, \text{plus2}, \text{plus3}\}.
\end{aligned}
$$

See Fig. 16.

The following specification is inductive:

$$\Theta_1^{\text{plus}}\text{: (ground plus1} \wedge \text{ground plus2)} \Rightarrow \text{ground plus3}.$$

In order to shorten this kind of specification we use arrows $\downarrow$ (to indicate that some argument is ground by hypothesis—input argument)

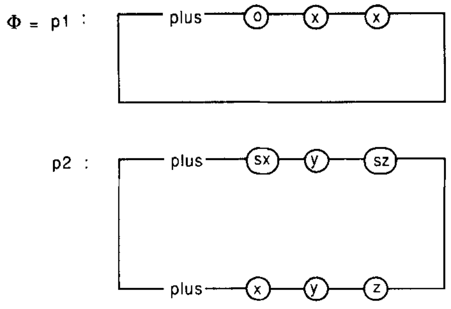

FIGURE 16

and ↑ (to indicate that some argument is ground; this the conclusion—output argument). Hence Θ_1^{plus} can be shortly written as ↓ ↓ ↑.

It is very easy to prove that the following specifications are inductive also:

$$\Theta_2^{\text{plus}}: \uparrow \uparrow \downarrow$$
$$\Theta_3^{\text{plus}}: \uparrow \downarrow \uparrow.$$

Θ_2^{plus} means that for every atomic goal of the form plus $(X, Y, s^n o)$ in which the third argument is given, all answer substitutions (if any) are ground.

This kind of specification called "mode" in PROLOG's folklore is useful for methodological and optimization purposes (see (Deransart and Maluszynski, 1984, 1985) for more details). It is not always inductive as shown by the following example:

(3.5.1) EXAMPLE (Deransart, 1984).

$$p(A, B) \Leftarrow q(A \cdot X, Y, Y, B \cdot X)$$
$$q(X, Y, a \cdot b, Y) \Leftarrow$$
$$q(X, Y, X, a \cdot b) \Leftarrow$$

In this program, a and b are constants and "·" is a binary operator.

All the proof tree roots of this program are such that $p2$ is ground. As such it could be a part of a bigger logic program.

It is partially correct w.r.t. the specification Θ where Θ^p is ↓ ↑ and Θ^q is ↓ ↓ ↑ ↑, but this specification is not inductive. Denoting by $p1$, $p2$ (resp. $q1 \cdots q4$) the arguments of p (resp. q), it is easy to show that the following specification is inductive:

Θ_i^p: ground $p2$.

Θ_i^q: $[(\text{ground } q1 \Rightarrow \text{ground } q3) \wedge \text{ground } q4]$

$\vee\ [(\text{ground } q2 \Rightarrow \text{ground } q4) \wedge \text{ground } q3]$.

And the following formulas hold also:

$$\Theta_i^p \Rightarrow \Theta^p$$

$$\Theta_i^q \Rightarrow \Theta^q.$$

(These results hold in any interpretation). Hence Θ is valid by Theorem (3.2.5).

Note that the difficulty in this example comes from the form of the inductive specification which is not obvious and thus difficult to find. The method presented in (4) can be viewed as an aid to find a demonstration and the results obtained in Section (5) as a way to find a stronger inductive specification. Automatization of the proof and automatic detection of modes have been considered in (Deransart and Maluszynski, 1984, 1985).

(4) Annotations

The practical usability of the proof method of Theorem (3.2.5) suffers from its theoretical simplicity: the inductive specification Θ' to be found to prove the validity of some given specification Θ will need complex formulas Θ'^X since there is *only one* for each X in N.

In what follows, we want to describe at a syntactical and abstract level a method to break such complex formulas into boolean combinations of simpler ones.

Roughly speaking, we shall write a formula Θ^X as $\mathbb{AND}\ A \Rightarrow \mathbb{AND}\ B$ where A and B are finite sets of formulas. The formulas in A will be considered as inherited attributes and those in B as synthesized ones within a certain attribute dependency scheme. The truth of $\mathbb{AND}\ A \Rightarrow \mathbb{AND}\ B$ corresponds to the fact that synthesized attributes depend on inherited ones (via the subtree issued from the node). And the non-circularity of the attribute dependency scheme will ensure the non-circularity of the proof of $\mathbb{AND}\ A \Rightarrow \mathbb{AND}\ B$ and hence its truth.

This method will be well-suited to (usual) attribute grammars where, at each node of the tree, some attribute occurrences w_0 are defined (in a unique way) in terms of neighbour attribute occurrences $w_1, ..., w_k$. In this case, $\mathbb{AND}\ B$ will contain some information concerning w_0, and $\mathbb{AND}\ A$ will contain some information concerning the values of $w_1, ..., w_k$ (the attribute occurrences at this node, upon which w_0 depends, via the subtree or the context).

In other words, the design of the formulas Θ^X can be made in connection with the structure of the dependencies between attribute occurrences. Certain restrictions defining subclasses of attribute grammars (strongly non-circular, ℓ-ordered etc.) certainly yield specific types of formulas Θ^X (but we shall not explore this aspect here).

Such an approach has been taken by Katayama and Hoshino (1981) for a class of attribute grammars closely connected with the class of benign attribute grammars of Mayoh (1981).

(4.1) *Definition*

Let $G = \langle N, P, \mathrm{Attr}, \Phi, \mathbb{D} \rangle$ be a relational attribute grammar.

An *annotation* of G is a mapping Δ assigning to every X in N a finite set $\Delta(X)$ of formulas of $\mathscr{L}'(\mathrm{Attr}(X))$ where $\mathscr{L}'$ is some logical language containing $\mathscr{L}$.

The set $\Delta(X)$ is partitioned into two sets $I\Delta(X)$ (the set of *inherited assertions* of X) and $S\Delta(X)$ (the set of *synthesized assertions* of X).

The specification Θ_Δ *associated with* Δ is the one such that Θ_Δ^X is the formula

$$\mathbb{AND}\ I\Delta(X) \Rightarrow \mathbb{AND}\ S\Delta(X).$$

We say that Δ is *valid* (resp. *inductive*) if Θ_Δ has the same property.

It is clear that every specification Θ can be seen as the annotation Δ_Θ such that $S\Delta_\Theta(X) = \{\Theta^X\}$ and $I\Delta_\Theta(X) = \{\mathrm{true}\}$ and that Θ_{Δ_Θ} coincides with Θ.

We shall now give sufficient conditions ensuring the validity of an annotation.

The intuition behind these conditions is to formalize a new way to make proofs, different from the inductive method, but which can be viewed as a special case of it. Instead of a unique inductive proof scheme, one now has local proof schemes associated to each production, using the annotation Δ. These schemes are formalized by the so-called logical dependency schemes. They indicate which hypotheses can be used locally to prove all the annotations in output positions. These schemes are locally valid ones, that do not ensure that Θ_Δ is also valid. One needs one more condition which corresponds to the non-circularity of the corresponding logical dependency scheme. This means intuitively that there is no circularity in the proof.

Hence Θ_Δ is a valid specification if the logical dependency scheme associated with Δ is sound (local validity) and non-circular.

The inductive specification method is clearly a special case of this method since the annotation is reduced to a single formula associated with each non-terminal and the logical dependency scheme is then purely synthesized.

(4.2) *Attribute Dependency Schemes*

We have introduced attribute grammars without dependencies (1.4, 1.5, 1.6). We now introduce attribute grammars with pure dependencies between attributes and no semantics. These new formal objects are called attribute dependency schemes.

Let us begin with some definitions and notations concerning binary relations and graphs.

(4.2.1) *Relations and graphs.* By a *graph* we mean a finite directed graph formally defined as a pair $D = (A, R)$ consisting of a finite set of vertices A and a binary relation $R \subseteq A \times A$. If $(a, b) \in R$ this means that there exists an arc from a to b. When A is known from the context we shall identify D with R (and write D for (A, D)). We denote by D^+ the graph (A, R^+) where R^+ is the transitive closure of R. By the union of a family of graphs $(D_i)_{i \in I}$ with $D_i = (A_i, R_i)$ we mean the graph $D = (\bigcup\{A_i/i \in I\}, \bigcup\{R_i/i \in I\})$.

(4.2.2) Definition. An *attribute dependency scheme* (ADS) is a 4-tuple $\langle N, P, \text{Attr}, D\rangle$ such that:

(i) $\langle N, P\rangle$ *is an abstract context-free grammar,*

(ii) Attr is a finite set of attributes as in Definition (1.4.1) (but sorts are irrelevant here),

(iii) D is a mapping associating with every p in P a binary relation $D(p) \subseteq W(p) \times W(p)$ (where $W(p)$ is as in (1.4.1)). It is also considered as a graph with set of vertices $W(p)$. This graph will be denoted by $D(p)$ and will be called the *local dependency graph* of p.

We now define $D(t)$, the *global* dependency graph of some t in $M(P)$, built by "pasting" together the $D(p)$'s at all nodes of t.

(4.2.3) Definition: Dependency Graphs. Let $t \in M(P)$. We denote by $D(t)$ the graph with set of vertices $W(t)$ (same notation as in (1.4.2)) and call it the *dependency graph* of t:

$$D(t) = \bigcup\{w \cdot D(p)/w \in \text{Node}(t),\ p = \text{lab}_t(w)\},$$

where

$$w \cdot D(p) = (w \cdot W(p),\ \rightarrow_{w,p}),$$
$$w \cdot W(p) = \{a(wu)/a(u) \in W(p)\},$$

and

$$a(wu) \xrightarrow[w,p]{} b(wu') \qquad \text{iff} \quad (a(u), b(u')) \in D(p).$$

We say that an ADS as above is *non-circular* if, for all t in $M(P)$, $D(t)$ has no cycle (and *circular* otherwise).

Deciding whether an ADS is non-circular is a trivial extension of the usual non-circularity test for attribute grammars. See (Deransart, Jourdan, and Lorho, 1984) for practical methods and references on circularity tests.

The study of the $D(t)$'s necessitates further notations.

(4.2.4) *Notations.* We denote by $D^+(t)$ the transitive closure of $D(t)$, by $D_0^+(t)$ the restriction of $D^+(t)$ to $W_0(t)$. Since $W_0(t)$ is in bijection with $\mathrm{Attr}(\sigma(t))$ (by $a(0) \mapsto a$) we shall consider $D_0^+(t)$ as a binary relation on $\mathrm{Attr}(\sigma(t))$, i.e., on the attributes of the root of t.

Let $p \in P_{\langle X_1 \cdots X_n, X_0 \rangle}$. Let $r_i \subseteq \mathrm{Attr}(X_i) \times \mathrm{Attr}(X_i)$ for $i = 1, ..., n$. We denote by $D(p)[r_1, ..., r_n]$ the graph $D(p) \cup 1 \cdot r_1 \cup \cdots \cup n \cdot r_n$ where $i \cdot r_i$ is the graph with set of vertices $\{a(i)/a \in \mathrm{Attr}(X_i)\}$ and set of arcs $\{(a(i), b(i))/(a, b) \in r_i\}$. We denote by $D_0^+(p)[r_1, ..., r_n]$ the restriction to $W_0(p) = \{a(0)/a \in \mathrm{Attr}(X_0)\}$ of $(D(p)[r_1, ..., r_n])^+$. Through the bijection $a \mapsto a(0)$ of $\mathrm{Attr}(X_0) \to W_0(p)$ we consider $D_0^+(p)[r_1, ..., r_n]$ as a binary relation on $\mathrm{Attr}(X_0)$.

(4.3) *Soundness of an Annotation*

Let G and Δ be as above. Let D be mapping associating with p in $P_{\langle X_1 \cdots X_n, X_0 \rangle}$ a graph $D(p)$ with set of nodes

$$V(p) = \{\phi(i) \mid 0 \leq i \leq n, \phi \in \Delta(X_i)\}$$

(i.e., $V(p)$ is defined w.r.t Δ as $W(p)$ is w.r.t. Attr) such that the target of every arc belongs to

$$V_{\mathrm{concl}}(p) = \{\phi(i) \in V(p) \mid \phi \in S\Delta(X_0) \text{ and } i = 0 \text{ or } \phi \in I\Delta(X_i) \text{ and } l \leq i \leq n\}.$$

Letting $\underline{\Delta}$ denote $\bigcup\{\Delta(X)/X \in N\}$ this means that $\langle N, P, \underline{\Delta}, D \rangle$ is an attribute dependency scheme; let us call it a *logical dependency scheme* (LDS). We shall be interested in non-circular logical dependency schemes.

We say that Δ is *D-sound* if for all p in P (of profile $\langle X_1 \cdots X_n, X_0 \rangle$), for all $\phi(i)$ in $V_{\mathrm{concl}}(p)$, the following condition holds,

$$\mathbb{D} \models [\Phi_p \;\textbf{and}\; \mathbb{AND}\; \{\psi_j \mid \psi(j) \in V(p), (\psi(j), \phi(i)) \in D(p)\}] \Rightarrow \phi_i$$

(where the mapping $\psi \mapsto \psi_j$ is as in Definition (3.2.1)).

(4.4) *The Use of Annotations for Correctness Proofs*

We shall prove that if Δ is D-sound and the associated LDS is not circular then Θ_Δ is valid. We give here a direct proof of this fact as in (Deransart, 1983). In Section 5 we will give another proof of this result after comparing annotations and fix-point induction.

(4.4.1) THEOREM. *Given an annotation Δ of a relational grammar $\langle N, P, \text{Attr}, \Phi, \mathbb{D}\rangle$ and a logical dependency scheme $\text{LDS}_\Delta = \langle N, P, \underline{\Delta}, D\rangle$, if Δ is D-sound and LDS_Δ is non-circular, then Θ_Δ is valid.*

Proof. The theorem is proved by induction, following the sketch given in (Deransart, 1983) for logical attribute grammars.

We prove that in every tree t of $M(P)_X$, if $\mathbb{AND}\{\varphi_0/\varphi \in I\Delta(X)\}$ holds, then all formulas φ_u, $u \in \text{Node}(t)$, $\varphi \in \Delta(\text{lab}_t(u))$ also hold; hence $\mathbb{AND}\{\varphi_0/\varphi \in S\Delta(X)\}$ holds. It follows that Θ_Δ^X is valid in $\mathbb{D}$ for all X in N; hence Θ_Δ is a valid specification.

If LDS_Δ is non-circular, then the relation $D(t)$ is a partial order (antisymetry holds since $D(t)$ has no cycle). The induction follows this partial order. By the construction of $D(t)$, its minimal elements are the formulas φ_0, $\varphi \in I\Delta(X_0)$, or φ_u which are instances of some formula in $V_{\text{concl}}(p)$ for some production p. As Δ is D-sound, these formulas are valid in $\mathbb{D}$ since they are instances of local proof schemes valid in $\mathbb{D}$.

Now consider any other formula in $D(t)$. By construction it is an instance of a formula of $V_{\text{concl}}(p)$ for some p. By construction, all formulas that it depends on in $D(t)$ are instances of formulas in $V(p)$. By induction hypothesis, all the formulas that it depends on hold in $\mathbb{D}$. As Δ is D-sound it holds also. Hence all formulas hold in $D(t)$.

(4.4.2) EXAMPLES. Continuing Examples (1.5.1) and (1.4.3) we reconsider the three given examples using the new method.

The Katayama–Hoshino example (1981) has been introduced to show the feasibility of such a method. They show the validity of the specification

$$\Theta \text{ defined as: } \exists n \quad k = 4 * n$$

using the same kind of method but requiring restrictions on the attribute grammar and the specification. We get here the same result without any specific hypothesis by using a sound and non-circular logical dependency scheme.

Consider the same annotation as in (Katayama and Hoshino, 1981) that is given by

$$\Delta = \{\alpha, \beta, \gamma, \delta\}$$

$I\Delta(S) = \phi,$	$S\Delta(S) = \{\delta\}$
$I\Delta(A) = \{\beta\},$	$S\Delta(A) = \{\alpha, \gamma\}$
$\alpha^A:$	$\exists n, g = f + 2 * n$
$\beta^A:$	$h = 2 * f + g$
$\gamma^A:$	$k = f + 2 * g$
$\delta^S:$	$\exists n, k = 4 * n.$

The logical dependency scheme is $\langle N, P, \underline{\Delta}, D\rangle$ (see Fig. 17).

The proofs given in (Katayama and Hoshino, 1981) show the soundness of LDS$_\Delta$. As it is obviously non-circular, Θ_Δ is valid, and hence δ^S and $\beta^A \Rightarrow \alpha^A \wedge \gamma^A$ hold.

This latter specification (Θ_Δ^A) does not seem very natural and the associated inductive stronger specification (as it is shown in Section 5, it is α^A **and** $(\beta^A \Rightarrow \gamma^A)$) is not a natural one either. The other (and stronger) one given at (3.2.6) seems to be more natural.

The second example illustrates the way a proof can be modularized in order to be made clearer. The inductive Θ (of (3.2.6)) can be broken into shorter formulas:

$$\Theta^S: \theta_1^S \textbf{ and } \theta_2^S$$

with

$$\theta_1^S: \textbf{baword}(x^S) \textbf{ or } \textbf{bword}(x^S)$$

$$\theta_2^S: x^S = y^S$$

$$\Theta^A: \theta_3^A \textbf{ and } \theta_4^A \textbf{ and } \theta_5^A$$

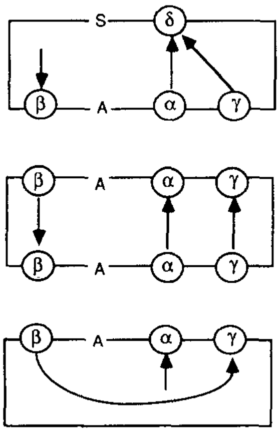

FIGURE 17

with

$$\theta_3^A : \mathbf{baword}(y^A) \text{ or } \mathbf{bword}(y^A)$$

$$\theta_4^A : \mathbf{baword}(u^A) \Rightarrow x^A = u^A$$

$$\theta_5^A : \mathbf{bword}(u^A) \Rightarrow x^A = u^A y^A.$$

Hence it can be easily observed that the proof of Θ^S in the first production can be split into separate proofs of θ_1^S and of θ_2^S, each of them using some properties of θ^A (i.e., some hypothesis θ_3^A, θ_4^A, or θ_5^A). For example, the implication Φ_{p_1} **and** $\theta_3^A \Rightarrow \theta_1^S$ holds trivially. Thus there is a new way to present such a proof by giving the logical dependency schemes associated with each production and proving their soundness and non-circularity. After giving the following logical dependency scheme, the proof becomes a sequence of very short and independent proofs (see Fig. 1, 2, and 18).

$$\Delta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$$

$$I\Delta = \phi$$

$$S\Delta(S) = \{\theta_1, \theta_2\}, \qquad S\Delta(A) = \{\theta_3, \theta_4, \theta_5\}.$$

The associated LDS is trivially non-circular since it is a purely synthesized one.

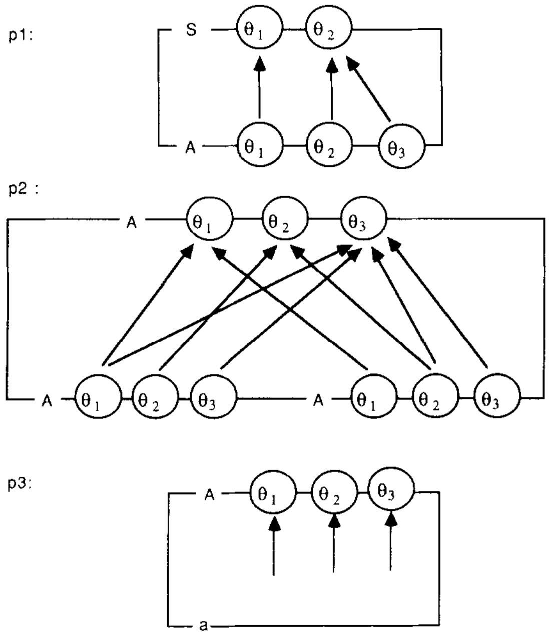

FIGURE 18

Let us consider again the logic program of Example (3.5.1) given in an attribute grammar style (see Fig. 19). Let us analyse locally what is necessary in order to prove that some argument is ground. For example, in production 1, $A \cdot X$ (first argument of q) is ground if A and $B \cdot X$ are, or in production 2 the fourth argument of q is ground if the second is. We consider only the necessary hypotheses.

Now we are able to build an annotation, recalling that the purpose of such a proof is to establish the validity of the mode $\downarrow \uparrow$ of p and $\downarrow \downarrow \uparrow \uparrow$ of q,

$$\Delta = \{g1p, g2p, g1q, g2q, g3q, g4q\}$$

$$I\Delta(p) = \{g1p\} \qquad I\Delta(p) = \{g2p\}$$

$$I\Delta(q) = \{g1q, g2q\} \qquad S\Delta(q) = \{g3q, g4q\},$$

where gir means "the ith argument of r is ground."

Thus it is easy to build in each production a sound LDS (see Fig. 20). It remains to show the non-circularity of the LDS to achieve the validity proof of Δ, and hence of the modes.

It is worth to noting that whether the formulas "g" are inherited or synthesized has no connection with the non-oriented "attributes" of the logic program. No orientation has been supposed concerning the predicate arguments. The fact that the same number of formulas is associated with

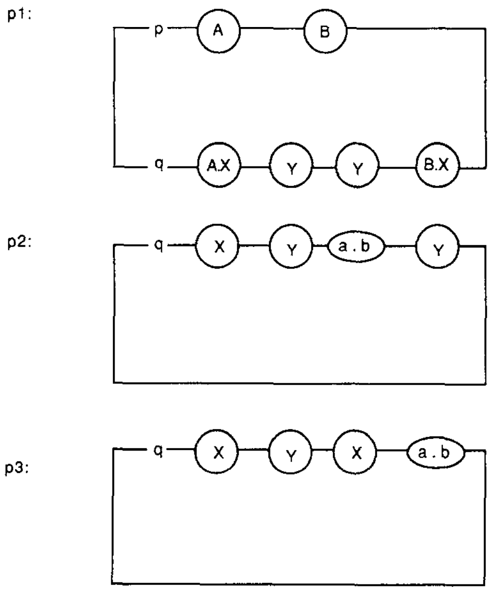

FIGURE 19

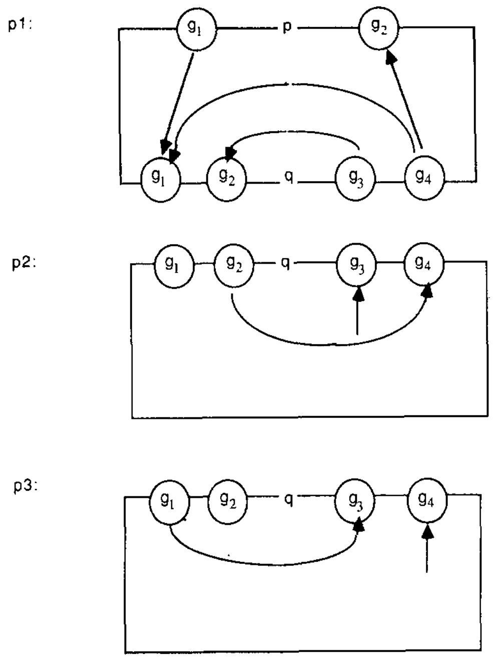

FIGURE 20

each grammatical sort comes from the fact that they are properties of one argument only.

Remark finally that a direct proof of validity of the modes is difficult since the corresponding inductive specification (Θ_i, see (3.5.1)) is not easy to formulate. In general the main difficulty is to prove the non-circularity of the LDS, which is decidable and can be performed automatically (Deransart, Jourdan, and Lorho, 1984).

(5) COMPARING ANNOTATIONS AND FIX-POINT INDUCTION

We shall show that if Δ is D-sound and LDS$_\Delta$ is non-circular, then there exists an inductive specification, say Θ', stronger than Θ_Δ, i.e., such that $\mathbb{D} \models \Theta' \Rightarrow \Theta_\Delta$, and that is written in the logical language where Δ is defined (provided this language is closed under the propositional connectives **and**, **or**, **not**).

This strengthens Theorem (4.4.1) and proves that the two methods of Sections 3 and 4 are equally powerful.

Dependency indicators will be introduced now to analyse properties of ADS's (and hence of LDS's). They are a way to summarize the possible dependencies between attributes of a given non-terminal coming from the possible subtrees. As the number of possible relations is finite, they are finite sets of relations. However, this number can be very large (it is in fact of exponential size). It has been proved (see (Deransart, Jourdan, and Lorho, 1985) for references) that simpler sets can be used to decide specific properties of attribute dependency schemes. We present here another way to establish the above results, which will be useful to show the relationships between annotations and fix-point induction.

We suppose an ADS of the form $\langle N, P, \text{Attr}, D\rangle$ is given.

(5.1.1) DEFINITION. We now introduce mappings $\mathscr{D}$ such that $\mathscr{D}(X)$ represents (or approximates in some sense) the set of $D_0^+(t)$'s for all t in $M(P)_X$, i.e., the set of possible dependencies at the root of trees in $M(P)_X$.

Let $\mathscr{D}$ be a mapping associating with every X in N a finite set $\mathscr{D}(X)$ of binary relations on $\text{Attr}(X)$. Such a mapping is called a *dependency indicator.*

We say that $\mathscr{D}$ is *non-circular* iff, for all p in $P_{\langle X_1 \cdots X_n, X_0\rangle}$, all r_1 in $\mathscr{D}(X_1), ..., r_n$ in $\mathscr{D}(X_n)$, $D(p)[r_1, ..., r_n]$ has no cycle. It is *closed* if for every such $p, r_1, ..., r_n$, there exists r_0 in $\mathscr{D}(X_0)$ such that $D_0^+(p)[r_1, ..., r_n] \subseteq r_0$.

We shall compare two dependency indicators $\mathscr{D}$ and $\mathscr{D}'$ by letting $\mathscr{D} \leq \mathscr{D}'$ iff for every X in N, r in $\mathscr{D}(X)$ there exists r' in $\mathscr{D}'(X)$ such that $r \subseteq r'$.

Let us denote by $\mathscr{D}_0$ the dependency indicator such that $\mathscr{D}_0(X) = \{D_0^+(t)/t \in M(P)_X\}$.

(5.1.2) PROPOSITION. *Given an ADS of the form* $A = \langle N, P, \text{Attr}, D\rangle$,

(1) $\mathscr{D}_0$ *is closed. If* $\mathscr{D}$ *is any closed dependency indicator, then* $\mathscr{D}_0 \leq \mathscr{D}$.

(2) *The ADS A is non-circular iff* $\mathscr{D}_0$ *is non-circular iff there exists a closed and non-circular dependency indicator for it.*

Proof. (1) That $\mathscr{D}_0$ is closed is an easy consequence of the definition. It is easy to prove by induction on the structure of t that, for all t in $M(P)$, there exists r in $\mathscr{D}(\sigma(t))$ such that $D_0^+(t) \subseteq r$.

(2) Remark first that A is circular iff there exist $p, t_1, ..., t_n$, such that $D(p)[D_0^+(t_1), ..., D_0^+(t_n)]$ has cycles. Hence A is non-circular iff $\mathscr{D}_0$ is iff there exists for A some non-circular and closed dependency indicator, since if $\mathscr{D}$ is such then $\mathscr{D}_0 \leq \mathscr{D}$ whence $\mathscr{D}_0$ is noncircular. (This corresponds to the standard non-circularity test for attribute grammars—see (Deransart, Jourdan, and Lorho, 1984)).

To every conditional attribute grammar G corresponds an attribute dependency scheme $\text{ADS}_G = \langle N, P, \text{Attr}, D\rangle$ where $D(p)$ is the local dependency graph defined as usual (see (Courcelle and Franchi-Zannettacci, 1982)).

Certain known subclasses of attribute grammars can be characterized in terms of the properties of the corresponding dependency indicators.

(5.1.3) PROPOSITION. *An attribute dependency scheme is strongly non-circular* (Courcelle and Franchi-Zannettacci, 1982) *iff there exists a closed and non-circular dependency indicator $\mathscr{D}$ such that $\mathscr{D}(X)$ is singleton for all X in N.*

It follows from the definition of the dependency indicator and Definition 3.12 of Courcelle and Franchi-Zannettacci (1982).

Another example is the class of attribute grammars in normal form such that the dependency indicator $\mathscr{D}_1$ such that $\mathscr{D}_1(X) = \{\mathrm{Inh}(X) \times \mathrm{Syn}(X)\}$ (which is necessarily closed by the normal form hypothesis) is non-circular. This coincides with the class of one-visit attribute grammars (Engelfriet and File, 1981) also called one-sweep in (Engelfriet and File, 1982a, b, 1984).

(5.1.4) PROPOSITION. (Courcelle, 1984): *An attribute dependency scheme in normal form is one-sweep (equivalently one-visit) iff the dependency indicator $\mathscr{D}_1$ is non-circular.*

(5.2) *The Use of Annotations for Correctness Proofs*

We shall prove that if Δ is D-sound and LDS_Δ is non-circular then Θ_Δ is valid, since it is weaker than an inductive specification.

To do the proof we shall associate with Δ another specification Θ which is inductive and stronger than Θ_Δ. This is the purpose of the following definition.

(5.2.1) *Dependency indicators and specifications.* Let G, Δ be as above, and D also. Let $\mathscr{D}$ be a dependency indicator. Let $\Theta_{\mathscr{D}}$ be the specification associated with $\mathscr{D}$ as follows:

$$\Theta^X_{\mathscr{D}} \text{ is } \mathbb{OR}\ \{\Theta^X_r / r \in \mathscr{D}(X)\},$$

where

$$\Theta^X_r \text{ is } \mathbb{AND}\ \{\Theta^X_{r,\phi} / \phi \in S\Delta(X)\},$$

and

$$\Theta^X_{r,\phi} \text{ is } \mathbb{AND}\ \{\psi \in \Delta(X) \mid (\psi, \phi) \in r\} \Rightarrow \phi.$$

(5.2.2) PROPOSITION. (1) *If $\mathscr{D} \leq \mathscr{D}'$ then $\Theta_{\mathscr{D}} \Rightarrow \Theta_{\mathscr{D}'}$.*

(2) *If $\mathscr{D}$ is closed and non-circular then $\Theta_{\mathscr{D}}$ is inductive.*

Proof. (1) If $r \subseteq r'$ then $\Theta^X_{r,\phi} \Rightarrow \Theta^X_{r',\phi}$; hence $\Theta^X_r \Rightarrow \Theta^X_{r'}$. Since $\mathscr{D} \leq \mathscr{D}'$, if $\mathscr{D}(X) = \{r_1, ..., r_k\}$ there exists a subset $A = \{r'_1, ..., r'_{k'}\}$ of $\mathscr{D}'(X)$ such that for all $i = 1, ..., k$ there exists j in $1, ..., k'$ such that $r_i \subseteq r'_j$. Hence

$$\Theta^X_{\mathscr{D}} \Rightarrow \mathbb{OR}\ \{\Theta^X_{r'_j} / 1 \leq j \leq k\} \Rightarrow \Theta^X_{\mathscr{D}'}.$$

Hence we have shown that $\Theta_{\mathscr{D}} \Rightarrow \Theta_{\mathscr{D}'}$.

(2) We must show that for p in $P_{\langle X_1 \cdots X_n, X_0 \rangle}$ the following formula holds (i.e., is valid in $\mathbb{D}$; this terminology will be used in the sequel of the proof):

$$\Phi_p \textbf{ and } \Theta^{X_1}_{\mathscr{D},1} \textbf{ and} \cdots \Theta^{X_n}_{\mathscr{D},n} \Rightarrow \Theta^{X_0}_{\mathscr{D},0}.$$

It suffices to show that for all r_1 in $\mathscr{D}(X_1)$, ..., r_n in $\mathscr{D}(X_n)$ there exists r_0 in $\mathscr{D}(X_0)$ such that the formula

$$\Phi_p \textbf{ and } \Theta^{X_1}_{r_1,1} \textbf{ and} \cdots \textbf{and } \Theta^{X_n}_{r_n,n} \Rightarrow \Theta^{X_0}_{r_0,0}$$

holds. We show this by taking some r_0 such that $D_0^+(p)[r_1, ..., r_n] \subseteq r_0$ (since $\mathscr{D}$ is closed). We need only show that for all ϕ in $S\Delta(X_0)$, the formula

$$\Phi_p \textbf{ and } \Theta^{X_1}_{r_1,1} \textbf{ and} \cdots \textbf{and } \Theta^{X_n}_{r_n,n} \textbf{ and } \mathbb{AND}\ \{\psi_0 \mid (\psi, \phi) \in r_0\} \Rightarrow \phi_0$$

holds. Let us abbreviate into ψ the left-hand side of this implication.

Let $H = D(p)[r_1, ..., r_n]$. Let K be the set of vertices of H consisting of $\phi(0)$ and all its predecessors in H (i.e., the vertices of H from which there is a path to $\phi(0)$).

We want to show that

$$\psi \Rightarrow \eta_k \qquad (*)$$

for all $\eta(k)$ in K.

Since H has no cycle, it suffices to show that $(*)$ holds, provided it holds for the immediate predecessors of $\eta(k)$ in H.

We do this by considering several cases:

Case 1. $\eta(k) \in V_{\text{concl}}(p)$.

The result follow from the hypothesis that Δ is D-sound (see Definition (4.3)).

Case 2. $\eta(k) \notin V_{\text{concl}}(p)$, $k = 0$. Hence $\eta \in I\Delta(X_0)$.

In this case $(\eta(0), \phi(0)) \in H^+$ implies $(\eta, \phi) \in r_0$; then $\psi \Rightarrow \eta_0$ holds in a trivial way since ψ is of the form ψ' **and** ψ_0 with $\psi_0 = \eta_0$.

Case 3. $\eta(k) \notin V_{\text{concl}}(p)$, $1 \leq k \leq n$. Hence $\eta \in S\Delta(X_k)$.

The immediate predecessors of $\eta(k)$ are the $\psi(k)$'s such that $(\psi, \eta) \in r_k$. The result follows from the fact that ψ is of the form

$$\psi' \textbf{ and } \Theta^{X_k}_{r_k,\eta,k} \ (\mathbb{AND}\ \{\psi_k \in \Delta(X_k)/(\psi, \eta) \in r_k\} \Rightarrow \eta_k).$$

(5.2.3) THEOREM (Reformulation of Theorem (4.4.1)). *Let G be a relational attribute grammar. If an annotation Δ of G is D-sound for some non-circular logical dependency scheme* LDS_Δ, *it is valid for G.*

Proof. Let $\mathscr{D}$ be any closed and non-circular dependency indicator for LDS_Δ (there exists some since LDS_Δ is non-circular, in particular the canonical one $\mathscr{D}_0$, and possibly other simpler ones). By Proposition (5.2.2)-2 the specification $\Theta_{\mathscr{D}_0}$ is inductive, hence valid.

The specification Θ_Δ is clearly equivalent to $\Theta_{\mathscr{D}_1}$ associated with the dependency indicator $\mathscr{D}_1$ of Proposition (5.1.4). Then by Proposition (5.2.2)-1 and since $\mathscr{D}_0 \leq \mathscr{D}_1$, $\Theta_{\mathscr{D}_0} \Rightarrow \Theta_\Delta$ and hence Θ_Δ is valid (since $\Theta_{\mathscr{D}_0}$ is inductive and valid).

(5.2.4) *Remark.* The proof of Theorem (5.2.3) shows how an inductive specification $\Theta_{\mathscr{D}}$ can be used instead of a D-sound annotation (where D is non-circular), to prove some valid specification.

The specification $\Theta_{\mathscr{D}_0}$ is quite complex. Relatively simple specifications $\Theta_{\mathscr{D}}$ are obtained if $\mathscr{D}(X)$ is a singleton for all X. This is the case if the LDS is strongly non-circular (see Proposition (5.1.3), or if the LDS is one-sweep (5.1.4) and in this last case the inductive specification $\Theta_{\mathscr{D}_1}$ coincides with Θ_Δ.

Note that since $\mathscr{D}$ is not a singleton in general, the inductive assertions of $\Theta_{\mathscr{D}}$ can have a size which is exponential in the size of Δ, counted as $\Sigma\{\text{Card}(\Delta(X))/X \in N\}$. Hence the complete inductive proof on G is of "exponential complexity" whenever the verification of the soundness of Δ is of "linear complexity" in the size of $\langle N, P, \underline{\Delta}, D\rangle$. This shows precisely how the introduction of annotations decreases the complexity of the correctness proof.

(5.2.5) EXAMPLE. We define a relational attribute grammar, an annotation Δ which is D-sound but such that Θ_Δ is not inductive. In fact we only define D and display it as a set of dependency graphs in a classical way (see for instance (Courcelle and Franchi-Zannettacci, 1982)),

$$N = \{X, Y\},\ P = \{p, q, r\},\ p \in P_{\langle Y,X \rangle},\ q, r \in P_{\langle \varepsilon, Y \rangle}$$

$$I\Delta(X) = \{\alpha\},\ S\Delta(X) = \{\beta\},\ I\Delta(Y) = \{\gamma, \mu\},\ S\Delta(Y) = \{\eta, \delta\}.$$

$D(p)$, $D(q)$, $D(r)$ are shown in Fig. 21.

The corresponding attribute dependency scheme is non-circular but not strongly non-circular.

We assume that Δ is D-sound, i.e., that the following formulas are valid:

(1) Φ_p **and** $\delta \Rightarrow \beta$

(2) Φ_p **and** α **and** $\delta \Rightarrow \gamma$

(3) Φ_p **and** $\eta \Rightarrow \mu$

(4) $\Phi_q \Rightarrow \eta$

(5) Φ_q **and** $\mu \Rightarrow \delta$

(6) $\Phi_r \Rightarrow \delta$

(7) Φ_r **and** $\gamma \Rightarrow \eta$.

The specification $\Theta = \Theta_\Delta$ is valid but not inductive. Its formulas are

$$\Theta^X: \alpha \Rightarrow \beta$$

$$\Theta^Y: \gamma \textbf{ and } \mu \Rightarrow \eta \textbf{ and } \delta.$$

Saying that Θ_Δ is inductive would mean that the following holds:

(8) Φ_p **and** (γ **and** $\mu \Rightarrow \eta$ **and** δ) $\Rightarrow$ ($\alpha \Rightarrow \beta$)

(9) $\Phi_q \Rightarrow$ (γ **and** $\mu \Rightarrow \eta$ **and** δ)

(10) $\Phi_r \Rightarrow$ (γ **and** $\mu \Rightarrow \eta$ **and** δ).

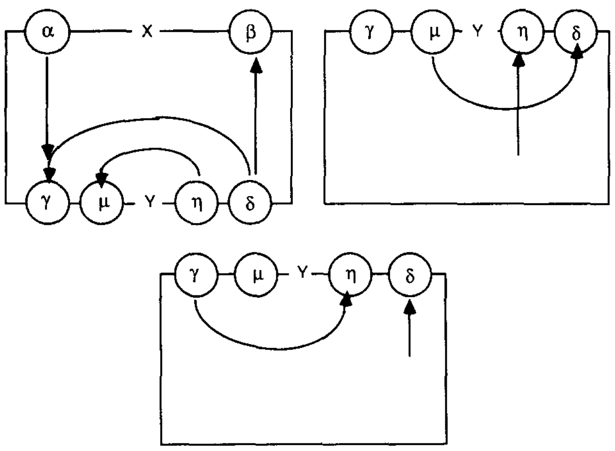

FIGURE 21

Whereas (9) and (10) easily follow from (4), (5), (6), and (7), (8) does not follow from (1)–(7).

It suffices to take Φ_p, α, η, μ equal to the constant boolean value **true** and Φ_q, Φ_r, β, γ, δ equal to **false** to verify this fact.

Here is the specification $\Theta' = \Theta_{\mathscr{D}_0}$,

$$\Theta'^X: \beta$$

$$\Theta'^Y: (\eta \textbf{ and } (\mu \Rightarrow \delta)) \textbf{ or } (\delta \textbf{ and } (\gamma \Rightarrow \eta)),$$

which is both inductive and stronger than Θ.

The reader will have noted that this example is isomorphic to the proof of the logic program in Example (4.4.2); hence the form of the inductive specification Θ_i in Example (3.5.1).

Acknowledgment

We thank the referees for many helpful comments and suggestions.

Received November 6, 1984; accepted March 5, 1987

References

Apt, K. R., and van Emden, M. H. (1982), Contributions to the theory of logic programming, *J. Assoc. Comput. Mach.* **29** (3), 841–862.

Cadiou, J. M. (1972), "Recursive Definition of Partial Functions and Their Computations," Ph. D. thesis, Stanford University.

Chirica, I. M., and Martin, D. F. (1979), An order-algebraic definition of knuthian semantics, *Math. Systems Theory* **13** (1), 1–27.

Clark, K. L. (1979), "Predicate Logic as a Computational Formalism," Res. mon. 79/56 TOC, Imperial College.

Courcelle, B. (1984), "Attribute Grammar: Definitions, Analysis of Dependencies, Proof Methods," Methods and Tools for Compiler Construction CEC-INRIA Course (B. Lorho, Ed.), Cambridge Univ. Press, Cambridge.

Courcelle, B., and Franchi-Zannettacci, P. (1982), Attribute grammars and recursive program schemes (I) and (II), *Theor. Comput. Sci.* **17** (2, 3), 163–191, 235–257.

De Bakker, J. W., and Meertens, L. G. L. T. (1975), On the completeness of the inductive assertion method, *J. Comput. System Sci.* **11** (3), 323–357.

Deransart, P. (1983), "Logical Attribute Grammars," IFIP'83, Paris (R. E. A. Mason, Ed.), North-Holland, Amsterdam, pp. 463–469.

Deransart, P. (1984), "Validation des Grammaires d'Attributs," Doctoral dissertation, Université Bordeaux I and INRIA, Rocquencourt.

Deransart P., and Ferrand G. (1987), An operational formal definition of PROLOG, RR INRIA 763.

Deransart, P., Jourdan, M., and Lorho, B. (1984), Speeding up circularity tests for attribute grammars, *Acta Inform.* **21**, 375–391.

DERANSART, P. JOURDAN, M., AND LORHO, B. (1985–86), "A Survey on Attribute Grammars," Parts I, II, III, INRIA, RR485, RR510, RR417; Lecture Notes in Computer Science, in press.

DERANSART, P., AND MALUSZYNSKI, J. (1984), "Modelling Data Dependencies in Logic Programs by Attribute Schemata," INRIA, RR323.

DERANSART, P., AND MALUSZYNSKI, J. (1985), Relating logic programs and attribute grammars, *J. Logic Programming* **2**, 119–155.

DUSKE *et al.* (1977), *IO*-macrolanguages and attributed translations, *Inform. and Control* **35**, 87–105.

ENGELFRIET, J. (1984), "Attribute Grammars: Attribute Evaluation Methods," Methods and Tools for Compiler Construction INRIA-CEC Course (B. Lorho, Ed.), pp. 103–108, Cambridge Univ. Press, Cambridge.

ENGELFRIET, J., AND FILE, G. (1981), The formal power of one-visit attribute grammars, *Acta Inform.* **16** (3), 275–302.

ENGELFRIET, J., AND FILE, G. (1982a), Simple multi-visit attribute grammars, *J. Comput. System Sci.* **24** (3), 283–314.

ENGELFRIET, J., AND FILE, G. (1982b), "Passes, Sweeps, and Visits in Attribute Grammars," memorandum INF-82-6, Onderafdeling der Informatica, Technische Hogeschool Twente; also Proc. of 8th ICALP, Lecture Notes in Computer Science Vol. 115, pp. 193–207.

FERRAND, G. (1987), Error Diagnosis in Logic Programming, an Adaption of E. Y. Shapiro's Methods, *J. Logic Programming* **4**, 177–198.

GALLIER, J. (1981), Nondeterministic flowchart programs with recursive procedures: Semantics and correctness, *Theoret. Comput. Sci.* **13**, 193–229, 239–270.

GOGUEN, J. A., THATCHER, J. W., WAGNER, E. G., AND WEIGHT, J. B. (1977), Initial algebra semantics and continuous algebras, *J. Assoc. Comput. Mach.* **24**, 68, 95.

GREIBACH, S. A. (1975), Theory of program structures schemes, semantics, verification, Lecture Notes in Computer Science Vol. 36, Springer-Verlag, Berlin/New York.

GUESSARIAN, I. (1981), Algebra semantics, Lecture Notes in Computer Science Vol. 99, Springer-Verlag, Berlin/New York.

KATAYAMA, T., AND HOSHINO, Y. (1981), Verification of attribute grammars, *in* "8th ACM POPL, Williamsburg, VA," pp. 177–186.

MANNA, Z. (1974), "Mathematical Theory of Computation," McGraw-Hill, New York.

MAYOH, B. H. (1981), Attribute grammars and mathematical semantics, *SIAM J. Comput.* **10** (3), 503–518.

MC CARTHY, J. (1965), "Lisp 1.5," MIT Press, Cambridge.

PAIR, C., AMIRCHAHY, M., AND NEEL, D. (1979), Correctness proofs of syntax-directed processing descriptions by attributes, *J. Comput. System. Sci.* **19** (1), 1–17.

VUILLEMIN, J. (1974), Correct and optimal implementations of recursion in a simple programming language, *J. Comput. System Sci.* **9**, 1–17.

VUILLEMIN, J. (1975), "Syntaxe Sèmantique et Axiomatique d'un Langage de Programmation Simple," Doctoral dissertation, Birkhauser Verlag.

WAND, M. (1978), A new incompleteness result for Hoare's system, *J. Assoc. Comput. Mach.* **25**, 168–175.

WATT, D. A, AND MADSEN, O. L. (1983), Extended attribute grammars, *Comput. J.* **26**, 142–153.