Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

E.A. van der Meulen

Deriving incremental implementations from algebraic specifications

Computer Science/Department of Software Technology Report CS-R9072 December

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Deriving Incremental Implementations from Algebraic Specifications

E.A. van der Meulen

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam
The Netherlands
(email: emma@cwi.nl)

We present a technique for deriving incremental implementations for a subclass of algebraic specifications, namely, conditional well-presented primitive recursive schemes. We use concepts of the translation of well-presented primitive recursive schemes to strongly non-circular attribute grammars, storing results of function applications and their parameters as attributes in an abstract syntax tree of the first argument of the function in question. An attribute dependency graph is used to guide incremental evaluation. The evaluation technique is based on a leftmost innermost rewrite strategy. The technique is extended to conditional well-presented primitive recursive schemes. Whereas in the non-conditional case attribute dependency graphs are constructed before evaluating a term, when working with conditional equations we construct the attribute dependency graph upon evaluation. The class of well-presented primitive recursive schemes is a very natural one for specifying the static semantics of languages. Allowing conditions to equations in a primitive recursive scheme is the first step in extending this class to one in which the dynamic semantics of languages can be described as well.

Key Words & Phrases: Software Engineering, Algebraic Specification, Conditional Term Rewriting, Incremental Evaluation, Attribute Grammars, Attribute Dependency Graphs.

1985 Mathematics Subject Classification: 68N20 [**Software**]: Compilers and generators; 68Q50 [**Theory of computing**]: Grammars, rewriting systems; 68Q65 [**Theory of computing**]: Abstract data types.

1987 CR Categories: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.6 [**Software Engineering**]: Programming Environments.

Note: Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments, phase 2 - GIPEII) and from the Netherlands Organization for Scientific Research - NWO, project *Incremental Program Generators*.

1. Introduction

In our quest for methods for deriving incremental implementations from algebraic specifications we inevitably came across incremental evaluators for attribute grammars. Courcelle and Franchi-Zannettacci proved that any well-presented primitive recursive scheme with parameters is equivalent to a strongly non-circular attribute grammar [CF82a, CF82b]. The parameters of a function are interpreted as the inherited attributes of a sort and the result of each function is interpreted as a synthesized attribute. Primitive recursive schemes, in turn, are a subset of algebraic specifications. Following this route we can transfer techniques developed for attribute grammars to algebraic specifications. In particular we can transfer techniques for incremental evaluation of attributes in a dependency graph to incremental evaluation of terms in an algebraic specification. Reps, Teitelbaum and Demers have described an optimal time algorithm for updating attribute values in the context of attribute grammars [RTD83]. We adapt their algorithm to an algorithm for updating attributes in the context of algebraic specifications.

We extend our technique to conditional well-presented primitive recursive schemes. In a non-conditional primitive recursive scheme each equation has a unique left-hand side. In a conditional primitive recursive scheme conditions concerning attributes or the abstract syntax tree can be added to the equations, and several equations may have the same left-hand side. Attribute dependencies are derived directly

from equations, So, if we use a non-conditional primitive recursive scheme for evaluating a term, an attribute dependency graph can be constructed before evaluation. If we use a conditional primitive recursive scheme the attribute dependency graph of a term is constructed upon evaluation. In this way, only the attribute dependencies of equations whose conditions succeed are added to the graph.

Our technique is to be implemented as part of the term rewriting engine of the ASF+SDF-system, a programming environment generator. From a language definition written in the algebraic specification formalism ASF+ SDF, a program environment is generated, at the moment consisting of a syntax directed editor and a term rewriting system for evaluating terms in the editor. We aim at changing as little as possible in the usual evaluation strategy of this system, which is a leftmost innermost rewrite strategy. Therefore, we do not want to use the full translation of algebraic specifications to attribute grammars but only the attribute concept of Courcelle and Franchi-Zannettacci and deduce attribute dependencies directly from a specification. Many algebraic specifications of static semantics of languages meet the requirements of a well-presented primitive recursive scheme. Conditional well-presented primitive recursive schemes provide more flexibility, however, and are a first step towards obtaining incremental implementations for a more comprehensive class of algebraic specifications.

The paper is organized as follows. In Section 2 we give definitions for attribute grammars, algebraic specifications, term rewriting strategies, well-presented primitive recursive schemes and the construction of an attribute grammar from a well-presented primitive recursive scheme. In Section 3 we describe how attributes and attribute dependency graphs can be constructed from a well-presented primitive recursive scheme, and the algorithms for first time evaluation of a term (i.e. evaluation of a term for which no abstract syntax tree has been stored yet), and incremental evaluation of a term are presented. Section 4 deals with the extension of these techniques to conditional primitive recursive schemes.

2. Definitions

2.1. Attribute grammars

An attribute grammar $<G,ATT,R,I>$ is a signature G, consisting of sorts and abstract tree constructors, extended with an attribute system. This means that for each sort X of G two disjoint sets of attributes are defined, the *inherited attributes* $INH(X)$ and the *synthesized attributes* $SYN(X)$. We also define INH and SYN

$$INH=\bigcup_X INH(X),\quad SYN=\bigcup_X SYN(X),\quad ATT=INH\cup SYN.$$

To each abstract tree constructor p of G semantic rules R_p are added for defining the value of the attributes of p. We define $R=\bigcup_{p\in G} R_p$. The interpretation I is the domain of attribute values.

For example, in Fig. 1 attributes and semantic rules are added to three constructors *program*,*empty-decls* and *decls* of the toy language ASPLE. Each ASPLE program consists of a, possibly empty, series of variable declarations followed by a series of statements. An interpretation is not given in this example but let us assume that the semantic functions describe the typechecking of an ASPLE program. The result of typechecking the declarations is a type-table: the synthesized attribute *tcdecls*. This provides the value of the inherited attribute for the statements, *env-stms*. The result (*true* or *false*) of typechecking a program equals the result of the typechecking of the statements, *tcstms*.

Attributes can be seen as labels attached to nodes in the abstract syntax tree used for transferring information through the tree. Roughly speaking, inherited attributes are used for transferring information down the tree, while synthesized attributes are used for transferring information up the tree. The semantic rules added to an abstract tree constructor define the values of both the inherited attributes of the children and the synthesized attributes of the parent. These attributes are called the *output attributes* of a constructor. They are defined in terms of the *input attributes* of the constructor: the inherited attributes of the parent and the synthesized attributes of the children.

An overview of attribute grammars is given in [DJL88]. Attribute grammars are widely used for defining the static semantics of programming languages, e.g, in the Cornell Synthesizer Generator [RT89b], the FNC-2 system [JP88], the GAG-system [Kas84] and the TOOLS system [KP90].

attributes:

$INH(PROGRAM)$	$= \varnothing$	$SYN(PROGRAM)$	$= \{tcp\}$
$INH(DECLS)$	$= \{env\text{–}decls\}$	$SYN(DECLS)$	$= \{tcdecls\}$
$INH(DECL)$	$= \{env\text{–}decl\}$	$SYN(DECL)$	$= \{tcdecl\}$
$INH(STMS)$	$= \{env\text{–}stms\}$	$SYN(STMS)$	$= \{tcstms\}$

constructors and semantic rules:

$$program : \text{DECLS} \times \text{STMS} \rightarrow \text{PROGRAM} \begin{cases} env\text{–}decls = empty\text{–}table \\ env\text{–}stms = tcdecls \\ tcp = tcstms \end{cases}$$

$$empty\text{–}decls: \rightarrow \text{DECLS} \quad tcdecls = env\text{–}decls$$

$$decls : \text{DECL} \times \text{DECLS}_1 \rightarrow \text{DECLS} \begin{cases} env\text{–}decl = env\text{–}decls \\ env\text{–}decls_1 = tcdecl \\ tcdecls = tcdecls_1 \end{cases}$$

Fig. 1. Part of the ASPLE typechecker in an attribute grammar formalism

2.2. Attribute dependencies

One of the major advantages of attribute grammars lies in the possibilities they offer for incremental evaluation of the attributes. By incremental evaluation we mean that if some attributes in an abstract syntax tree get a new value, for instance, because of a subtree replacement, we do not re-evaluate all other attributes but only those, whose value depend on the ones changed. For this we need a dependency graph of attributes in an abstract syntax tree. For each abstract tree constructor $p : X_1 \times \cdots \times X_n \rightarrow X_0$ an *attribute dependency graph* D_p can be defined as follows: the vertices of D_p are attributes of $X_0, \ldots, X_n$ and (a,b) is an edge of D_p if and only if a is used in the definition of b.

To obtain a dependency graph D_t of the attributes of a tree t of the grammar we take the union of all D_p of instances of constructors p in the tree. An attribute grammar is *well-formed* or *non-circular* if D_t is cycle-free for each tree. In that case an evaluation order for attributes of each abstract syntax tree exists. The attribute grammar is called *strongly non-circular* if for each node in each tree an evaluation order of attributes exists that does not depend on the particular subtree rooted at that node. In Fig. 2 part of an attribute grammar is shown that is not strongly non-circular, as the evaluation order of the attributes of X is determined by the constructor applied at X.

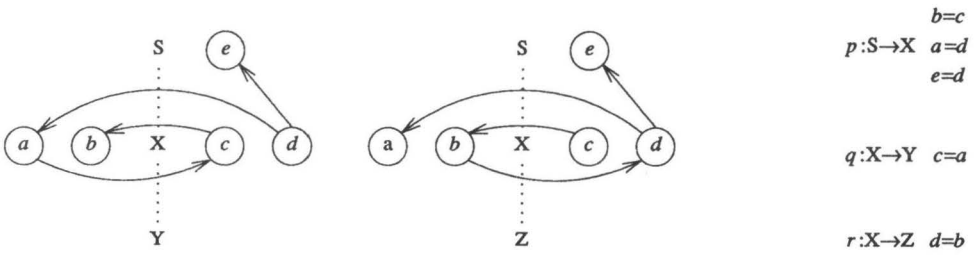

Fig. 2. A non-circular attribute grammar which is not strongly non-circular

2.3. Algebraic specifications

An algebraic specification consists of a signature and a set of (conditional) equations. In the signature sorts and functions over the sorts are defined. The equations define equalities between the terms that can be constructed from the signature. Algebraic specifications can be given of any abstract data-type, and can be used to describe the syntax and static and dynamic semantics of languages. Several formalisms for defining algebraic specifications exist. In the example in Fig. 3 part of the ASPLE typechecker is written in the algebraic specification formalism ASF [BHK89, Meu88]. Algebraic specifications are usually implemented as term rewriting systems. Equations are considered as rewrite rules, with an orientation from left to right. Evaluating a term means reducing it as far as possible. The result of such a reduction is a *normal form* of a term.

functions:
program : DECLS × STMS → PROGRAM
empty-decls: → DECLS
decls : DECL × DECLS → DECLS

tcp : PROGRAM → BOOL
tcdecls : DECLS × ENV → ENV
tcdecl : DECL × ENV → ENV
tcstms : STMS × ENV → BOOL

variables:
Decls :→DECLS Decl :→DECL
Env_{DECLS} :→ENV Stms :→STMS

equations:
[1] *tcp*(*program*(Decls,Stms)) = *tcstms*(Stms, *tcdecls*(Decls, empty-env))
[2] *tcdecls*(*empty-decls*, Env_{DECLS}) = Env_{DECLS}
[3] *tcdecls*(*decls*(Decl, Decls), Env_{DECLS}) = *tcdecls*(Decls, *tcdecl*(Decl, Env_{DECLS}))

Fig. 3. Part of the ASPLE typechecker in ASF

The idea of incremental evaluation of terms is that in re-evaluating a modified term we make use of the normal forms of terms already found and we re-evaluate as little as possible. The naïve way of doing this is to store all terms that occur during the reduction process together with their normal form. This would, obviously, take too much storage, and require long searches to determine if a term has been reduced before. The idea would be more practical if a smart way of storing (a selection of) terms could be found. This is too much to hope for in arbitrary rewrite systems as successive terms in a reduction can be very different.

Courcelle and Franchi-Zannettacci introduced a special kind of algebraic specifications: *Well-presented Primitive Recursive Schemes with parameters* (well-presented PRS, for short) [CF82a, CF82b], and they proved that a well-presented primitive recursive scheme is isomorphic to a strongly non-circular attribute grammar. From the viewpoint of algebraic specifications, well-presented primitive recursive schemes provide a structure that makes it much more obvious how to select the terms whose normal forms are to be stored, and how to store them economically.

2.4. Primitive recursive schemes

The following is a brief description of the results of Courcelle and Franchi-Zannettacci. First, we list the properties that make an algebraic specification a PRS. Next, we describe the basics of the construction of an attribute grammar from a PRS. Then, we explain why we need well-presentedness and we define this notion.

(i) A PRS is indicated as a triple $<G,S,\Phi>$, with G and S two signatures with disjoint sets of sorts, and Φ a set of functions. The signature of a PRS is the union of G, S and the signature of Φ. There are no equations involving only G-terms. Intuitively, G represents a set of tree constructor functions, describing, for instance, the abstract syntax of a programming language. For elements of Φ the following holds.

(ii) The type of the first argument of each ϕ in Φ is a sort from G and the types of all other arguments, called the *parameters of* ϕ, and the type of the output sort are sorts of S. Φ_X is the set of all functions of Φ that have the sort X as first argument. We assume function names in Φ not to be overloaded.

(iii) For each abstract tree constructor $p:X_1\times\cdots\times X_n\rightarrow X_0$ in G and each function ϕ in Φ_{X_0} exactly one defining equation $eq_{\phi p}$ exists:

$$\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau \qquad \textbf{(eq1)}$$

(iv) All x_i and y_j in the left-hand side of **(eq1)** are different variables.

(v) All defining equations are strictly decreasing in G: in **(eq1)** the only terms of G that are allowed in τ are the variables $x_1,\ldots,x_n$. So, τ is a $S\cup\Phi\cup\{x_1,\ldots,x_n\}\cup\{y_1,\ldots,y_m\}$-term

Some more definitions: A *G-term* is a term of some type $X\in G$, an *S-term* is a term of type $S\in S$, and we use *Φ-term* to indicate a term of which the head symbol is a function of Φ. All Φ-terms are *S*-terms. The parameters of a function ϕ are indicated by $PAR_\phi=\{par(\phi,j)\mid 1\leq j\leq arity(\phi)-1\}$. In **(eq1)** the parameters of ϕ are represented by the variables $y_1,\ldots,y_m$.

In the construction of an attribute grammar Γ from a PRS $<G,S,\Phi>$, G is the signature of Γ and S the domain of the attribute values. $SYN(X)$ consists of all functions ϕ of Φ_X. $INH(X)$ is defined by the set of parameters of all ϕ in Φ_X. Because of property (iv), the first argument in a defining equation $eq_{\phi p}$ coincides with an abstract tree constructor $p:X_1\times\cdots\times X_n\rightarrow X_0$ in Γ. Such an equation is used to deduce semantic rules $R_{\phi p}$ for attributes of p. The set R_p of all semantic rules of p is the union over all $R_{\phi p}$ with $\phi\in\Phi_{X_0}$.

Example 1.

The specification in Fig. 3 is a part of a PRS: *program*,*empty–decls* and *decls* are abstract tree constructors in G, S is formed by the union of the signature of the Booleans and signature of the Type-environments, and Φ are the typecheck functions $\{tcp,tcdecls,tcstms\}$. The construction of synthesized attributes is obvious. We use variable names for the construction of the inherited attributes, assuming that Env_{STMS} is the variable used in the defining equation of *tcstms*. We derive from equation [1] in Fig. 3:

$$R_{tcp\ program}:\begin{cases}Env_{DECLS}=empty\text{–}env\\ Env_{STMS}=tcdecls\\ tcp=tcstms\end{cases}.$$

As *tcp* is the only element of $\Phi_{program}$, $R_{tcp\ program}$ is the whole set of rules $R_{program}$. From equations [2] and [3] in Fig. 3 we derive

$$R_{tcdecls\ empty\text{–}decls}:\ tcdecls=Env_{DECLS}$$

$$R_{tcdecls\ decls}:\begin{cases}Env_{DECL}=Env_{DECLS}\\ Env_{DECLS_1}=tcdecl\\ tdecls=tcdecls_1\end{cases}.$$

Thus, we obtain the attribute grammar of Fig. 1.

Example 2.

Consider the following equations

$$\phi(p(x_1,x_2),y)=\chi(x_1,f(y))\qquad \psi(p(x_1,x_2),z)=\chi(x_1,constant)$$

Let Inh_ϕ, Inh_ψ and Inh_χ be the inherited attributes corresponding to the parameters of ϕ , ψ and χ. If we derive $R_{\phi p}$ and $R_{\psi p}$ we find that both sets contain a rule for Inh_χ, namely: $Inh_\chi=f(Inh_\phi)$ and $Inh_\chi=constant$. This means we will not find a proper set of rules R_p by taking the union $R_{\phi p}\cup R_{\psi p}$.

Example 2 illustrates that we have to define a property for primitive recursive schemes which guarantees that inherited attributes of the corresponding attribute grammar are uniquely defined. We call this property *well-presentedness*. In the definition of well-presentedness requirements concerning the relation between variable names in different equations are mentioned. This may seem strange, but, on the one hand, we want to use variables to construct inherited attributes, as we did in Example 1, and this construction is straightforward if each parameter is represented by exactly one variable (vi). On the other hand, if

each parameter is represented by exactly one variable, a simple syntactic check decides whether inherited attributes (parameters) are uniquely defined (vii).
A PRS is *well-presented* if

(vi) For any two equations concerning the same function ϕ

$$\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau \quad \phi(q(u_1,\ldots,u_{n'}),z_1,\ldots,z_m)=\tau',$$

the variable lists $<y_1,\ldots,y_m>$ and $<z_1,\ldots,z_m>$ are identical;

(vii) In all occurrences of the form

$$\cdots \psi(x_i,v_1,\ldots,v_m)\cdots$$

in right-hand sides of defining equations of the same abstract tree constructor p, corresponding parameter terms v_j are identical.

(viii) If two parameters $par(\psi,j)$ and $par(\xi,k)$ are represented by the same variable in left-hand sides of equations over the same constructor p, in all subterms

$$\psi(x_i,v_1,\ldots,v_m), \quad \xi(x_i,w_1,\ldots,w_{m'})$$

of right-hand sides of defining equations $eq_{\phi p}$ and $eq_{\psi p}$ v_j is identical to w_k.

Requirement (viii) states that it is allowed to represent different parameters by the same variable, if the parameters are defined in the same way. This is illustrated in the following example.

Example 3.

In a PRS with constructors $p:X_1\times X_2\rightarrow X_0$, $q:X_3\rightarrow X_1$ and equations

$$[1]\ \phi(p(x_1,x_2),y)=f(\psi(x_1,y),\chi(x_2,y))$$

$$[2]\ \psi(q(x_3),z)=\xi(x_3,f(z))$$

$$[3]\ \chi(q(x_3),z)=\xi(x_3,f(z))$$

the first equation says that $par(\psi,1)$, i.e. y, is equal to $par(\chi,1)$. Therefore, we may use one variable, z, to represent both parameters in the left-hand sides of the other equations. If we would not have done this $par(\xi,1)$ would have been defined in two different ways in [2] and [3] and the equations would not have been well-presented.

Of course, renaming of variables does not change the meaning of a specification, as long as no name clashes are introduced. What really matters is whether a PRS can be made well-presented by a suitable renaming of variables. In that case we call the PRS *well-presentable*. The PRS in Example 2 is neither well-presented nor well-presentable. In their paper, Courcelle and Franchi-Zannettacci have given an algorithm for deciding whether a PRS is well-presentable. The algorithm transforms a well-presentable PRS into a well-presented one.

3. Evaluation of terms

For incremental evaluation of terms in a PRS we consider functions and parameters of functions attributes of a tree. We will not bother with the full translation to attribute grammars, as we aim at finding an incremental evaluation technique for terms over a primitive recursive scheme which is as close as possible to the left-most innermost rewriting technique we already use for general algebraic specifications. The normal forms of Φ-terms are stored in the following way. The first argument of a given Φ-term is a G-term, say T. As no equations are given on terms of G, T is already in normal form. We store this term, or tree. The other direct subterms of a Φ-term are parameters of the function in the head symbol. We store the normal forms of these parameters as attributes of the top node of T. Finally, the normal form of the whole Φ-term is stored as an attribute of the top node of T as well. Likewise, we store the normal forms of all Φ-terms that are met during reduction. Since the PRS is strictly decreasing in G, all G-terms T' of these Φ-terms are subterms of the original G-term T. So, all normal forms of functions and parameters are stored as attributes of nodes in one tree.

3.1. Attribute dependencies in a primitive recursive scheme

A single modification in a term $\phi(T,t_1,\ldots,t_m)$ corresponds either to a subtree replacement in T or to a change in a parameter attribute at the top node of T. After such a modification the value of some parameter attributes may have changed and the value of some function attributes have to be recalculated. Also incremental evaluation can be used when a term $\psi(T,s_1,\ldots,s_k)$ has to be evaluated and we already have an attributed abstract syntax tree of T. We calculate new attribute values for ψ and its parameters and we may be able to use already obtained values of other attributes in T. In all cases we need to know which attributes may be affected when an attribute has changed value or which attributes are used for calculating other attributes. In other words, we need to know the attribute dependencies. Attribute dependencies for each abstract tree constructor in G can be deduced from a PRS. From each defining equation $eq_{\phi p}$ we derive dependencies $D_{\phi p}$ between function attributes and parameter attributes. $(a,b)\in D_p$ means that the value of a is needed to calculated the value of b. In the following derivation ϕ indicates both the function name and the corresponding attribute, $att(par(\phi,j))$ indicates the attribute that corresponds to the *j-th* parameter of ϕ. We derive $D_{\phi p}$ from the equation $\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau$:

- For each y_j in τ:
  - if an occurrence of y_j is a subterm of a Φ-term in τ, ϕ depends on $att(par(\phi,j))$.
  - If y_j occurs as the subterm of a Φ-term in τ, and $\psi(x_i,...v_k,...)$ is the smallest Φ-term that encloses y_j with y_j a subterm of v_k, then $att(par(\psi,k))$ depends on $att(par(\phi,j))$.
- For each Φ-term in τ e.g $\psi(x_i,v_1,\ldots,v_k)$:
  - if an occurrence of $\psi(x_i,v_1,\ldots,v_k)$ is not a subterm of a Φ-term τ, ϕ depends on ψ,
  - if $\psi(x_i,v_1,\ldots,v_k)$ occurs as subterm of a Φ-term in τ, and $\xi(x_i,...w_l,...)$ is the smallest Φ-term that encloses it, with $\psi(x_i,v_1,\ldots,v_k)$ a subterm of w_l, then $att(par(\xi,l))$ depends on ψ.

The dependencies that can be derived from the equations in Fig. 3 are:

$$D_{tcp\ program} = \{(tcdecls,\ Env_{\text{STMS}}),\ (tcp,tcstms)\}$$
$$D_{tcdecls\ empty-decls} = \{(Env_{\text{DECLS}},tcdecls)\}$$
$$D_{tcdecls\ decls} = \{(tcdecl,\ Env_{\text{DECLS}_1}),(tcdecls_1,tcdecls)\}$$

Fig. 4 shows the top of the attributed abstract syntax tree we obtain when the term $tcp(program(decls(int{:}x,empty\text{–}decls),stms(assign(x,5))))$ is reduced.

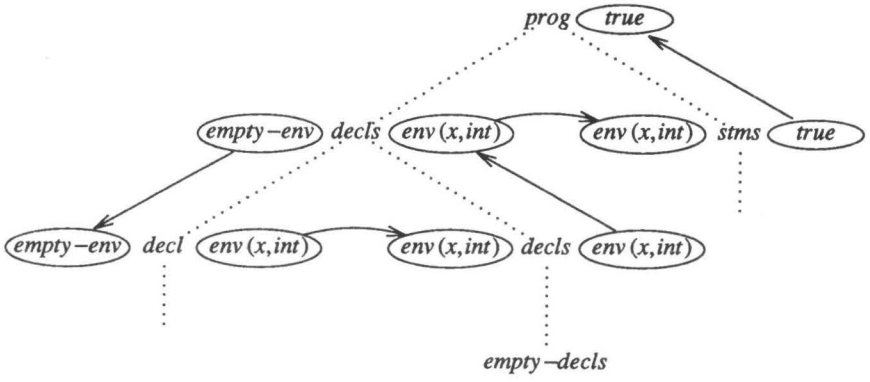

Fig. 4. Part of an attributed tree of an ASPLE program

The algorithm for deducing a dependency graph from an equation is given in Fig. 5. The dependency graph D_p of all attributes of an abstract tree constructor is the union of all $D_{\phi p}$. These dependencies are the same that exist in the isomorphic attribute grammar.

The attribute dependency graph D_T of a tree T, is obtained by pasting together all dependencies of the instances of constructors that occur in the tree. The graph D_T is non-circular for each T. This is an immediate consequence of the isomorphism proved by Courcelle and Franchi-Zannettacci. Informally speaking: It is clear that each $D_{\phi p}$ is non-circular. Due to the strictly decreasing property no dependencies

MAKEDEP–EQ$(eq_{\phi p})$
$eq_{\phi p}$ is $(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau$
$D_{\phi p}:=\varnothing$
MAKEDEP–FUNC$(\tau, eq_{\phi p})$
END

$D_{\phi p}$ is a global variable indicating the set of attribute dependencies.

MAKEDEP–FUNC(*functerm*,$eq_{\phi p}$)
functerm is a subterm of the right hand side of $eq_{\phi p}$
case *functerm* $= y_j$, $j\in\{1,\ldots,m\}$
  add $(att(par(\phi, j)), \phi)$ to $D_{\phi p}$
case *functerm* $= f(s_1,\ldots,s_k)$, $f\notin\Phi$
  for j=1 to k **do**
    MAKEDEP–FUNC$(s_j, eq_{\phi p})$ **od**
case *functerm* $= \psi(x_i, v_1, \ldots, v_k)$
  add (ψ, ϕ) to $D_{\phi p}$
  for j=1 to k **do**
    MAKEDEP–PAR$(v_j, \psi, j, eq_{\phi p})$ **od**
END

MAKEDEP–PAR(*parterm*, ψ, m, $eq_{\phi p}$)
parterm is a subterm of the term at the m-th parameter position of ψ in the right hand side of $eq_{\phi p}$
case *parterm* $= y_j$, $j\in\{1,\ldots,m\}$
  add $(att(par(\phi, j)), att(par(\psi, m)))$ to $D_{\phi p}$
case *parterm* $= f(s_1,\ldots,s_k)$
  for j=1 to k **do**
    MAKEDEP–PAR$(s_j, \psi, m, eq_{\phi p})$ **od**
case *parterm* $= \xi(x_l, v_1, \ldots, v_k)$
  add $(\xi, att(par(\psi, m)))$ to $D_{\phi p}$
  for j=1 to k **do**
    MAKEDEP–PAR$(v_j, \xi, j, eq_{\phi p})$ **od**
END

Fig. 5. Algorithms for deducing a dependency graph from an equation

between synthesized attributes of the top node of a constructor p can be derived from an equation $eq_{\phi p}$. It is also obvious that taking the union of all $D_{\phi p}$ to obtain D_p will not introduce cycles. Thus, we only have to check that the pasting of graphs D_p does not introduce cycles: The dependencies between attributes of a node N in a tree caused by the underlying subtree are always a subset of the set of dependencies between parameter attributes and the function attributes they belong to: $\{(att(par(\phi, k)), \phi) \mid \phi\in\Phi_N, 1\le k\le arity(\phi)-1\}$. As for the dependencies between attributes of a node N caused by the constructor p at the parent node of N, we note that well-presentedness guarantees for any pair ϕ , ψ of function attributes at N, that if a parameter of ψ depends (directly or transitively) on ϕ in D_p, no parameter of ϕ will depend on ψ. This means a parameter attribute never depends on a function attribute it belongs to.

3.2. First time evaluation

If a Φ-term $\phi(T, t_1,\ldots,t_n)$ has to be evaluated using a well-presented PRS, the abstract syntax tree of T is attributed with attributes with value "unknown" and a dependency graph D_T is constructed using the dependencies deduced for each abstract tree constructor. The reduction strategy we use is the *leftmost innermost strategy*, that is: if a term has to be reduced we first reduce all its subterms, starting with the left-most subterm. Reducing these subterms is done in the same way. While reducing, normal forms of Φ-terms and of parameter terms are stored in attributes. The algorithms in Fig. 6 and Fig. 7 describe innermost reduction interleaved with checks on the value of an attribute: each time a Φ-term or a parameter-term has to be reduced the corresponding attribute is checked. If this attribute already contains a value, reduction of the term is skipped and the term is replaced by this value. Otherwise, the term is reduced to its normal form, and the result is stored in the attribute.

REDUCE–NEW(*term*,T)
term is a closed Φ-term:
$\phi(p(x_1,\ldots,x_n),t_1,\ldots,t_m)$
for j=1 to m **do**
  t_j := normal form of t_i
  store t_j in $att(par(\phi, j))$ at T **od**
term matches with equation
$\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau$
$value_\phi$:= *REDUCE–FUNC*$(\tau, eq_{\phi p}, T)$
END

T=attributed parse tree of $p(x_1,\ldots,x_n)$
T' is a subtree of T, T'_i is the i-th subtree of T'
a, b, ϕ, ψ are attributes,
initially all attributes have value "unknown"

Fig. 6. Algorithm for reducing a Φ-term and storing values in attributes

```
REDUCE–FUNC(functerm, eq_φp, T')
functerm is (a subterm of) the right hand side
of eq_φp
case functerm = parameter of φ with attribute a in T'
   return value_a
case functerm = f(s_1, ..., s_k), f ∉ Φ
  for j = 1 to k do
    s_j := REDUCE–FUNC(s_j, eq_φp, T') od
  return normal form of f(s_1, ..., s_k)
case functerm = ψ(x_i, v_1, ..., v_k)
  if value_ψ = "unknown" at T'_i then
    x_i := T'_i
    for j = 1 to k do
      let b be the attribute for par(ψ, j) at T'_i
      if value_b = "unknown" then
        value_b := REDUCE–PAR(v_j, ψ, j, eq_φp, T') fi
      v_j := value_b od
    The instantiated functerm matches with equation
    eq_ψq with right hand side σ
    value_ψ := REDUCE–FUNC(σ, eq_ψq, T'_i) fi
  return value_ψ
END
```

```
REDUCE–PAR(parterm, ξ, m, eq_φp, T')
parterm is (a subterm of) the term
at the m-th parameter position of ξ
in the right hand side of eq_φp
case parterm = parameter of φ with attribute a in T'
   return value_a
case parterm = f(s_1,...,s_k), f ∉ Φ
  for j=1 to k do
    s_j := REDUCE–PAR(s_j, ξ, m, eq_φp, T') od
  return normal form of parterm
case parterm = ψ(x_i, v_1, ..., v_k)
  if value_ψ = "unknown" at T'_i then
    x_i := T'_i
    for j = 1 to k do
      let b be the attribute of par(ψ, j) at T'_i
      if value_b = "unknown" then
        value_b := REDUCE–PAR(v_j, ψ, j, eq_φp, T') fi
      v_j := value_b od
    The instantiated parterm matches with equation
    eq_ψq with right hand side σ
    value_ψ := REDUCE–FUNC(σ, eq_ψq, T'_i) fi
  return value_ψ
END
```

Fig. 7. Algorithms for reducing function-terms and parameter-terms

3.3. Incremental evaluation

To describe incremental evaluation we consider the case in which a term $\phi(T, t_1, \ldots, t_n)$ is offered for evaluation after the term $\phi(T', t_1, \ldots, t_n)$ has already been reduced, and T differs from T' in a single subtree. In other words: T is obtained from T' by replacing *oldsub* by *newsub* in node N. All functions on *newsub* have to be evaluated. If the new values of function attributes of N differ from the old values, other function attributes, depending on the ones changed, have to be re-evaluated.

Two considerations are at the basis of the design of the algorithm for incremental evaluation: Firstly, we want to re-evaluate values of attributes (parameters or functions) only if the value of a predecessor attribute has changed. Secondly, we want to evaluate each attribute at most once. For this purpose each attribute is given an *index* to indicate its status: "*C*" (changed), "*UC*" (unchanged) or "*TE*" (to be evaluated). To avoid double evaluation of attributes we only evaluate a function attribute after its parameters have obtained their final value. Another thing to keep in mind is that, unlike attribute grammars, primitive recursive schemes only provide defining equations for function attributes. The definition for parameter attributes is not given explicitly. This means that if some function attribute of a node N has got a new value and its successor happens to be a parameter attribute of N or of one of its siblings, we have to re-evaluate a function attribute of the parent node to find the new value of this parameter attribute.

When we start evaluating the functions on the new subtree, we have to know the evaluation order of the functions. The order of evaluation of functions of a node N in a tree T is determined by the *context* of N, i.e. T minus the tree rooted at N. It does not depend on the tree rooted at N. This follows directly from the fact that the graph we use is isomorphic to the one we would obtain for the corresponding the one we would obtain for the strongly non-circular attribute grammar corresponding to the PRS. It follows, as well, from the argumentation in Section 3.1.

We define a special case of the *superior characteristic graph* introduced by Reps, Teitelbaum and Demers [RTD83]. This graph describes for each node in a tree which parameter attributes depend, directly or transitively, on which function attributes. The algorithm *CREATE–SUP*(T, D_T), not explicitly given here, describes the top-down construction of all superior characteristic graphs in a tree. The root of a tree does not have such a graph, the construction of a superior characteristic graph of any other node in a tree is described in Fig. 8.

We use this superior characteristic graph to determine the evaluation order of functions at a node N. Function ϕ has to be evaluated before ψ if some parameter of ϕ depends on ψ: $(\psi, par(\phi,j)) \in D_N^{sup}$. Thus

we obtain a partial evaluation ordering of attributes in each node.

```
CREATE–SUP–NODE(N, T,D)
N^sup := ∅
for all function attributes ϕ of N do
  if an attribute a of N exists such that (ϕ, a)∈D
     add (ϕ, a) to N^sup fi
  if an attribute a of N exists
  and attributes ψ and b of the parentnode M of N
  such that: (ϕ, ψ)∈(D_T/D_{N–sub})^+
        (ψ, b)∈M^sup
        (b, a)∈(D_T/D_{N–sub})^+ then
    add (ϕ, a) to N^sup fi
od
END
```

N–sub is the subtree of T rooted at N
D_T is the attribute dependencygraph of T
D_T/D_{N-sub} is D_T minus the
dependencies between attributes of N–sub.
$(D_T/D_{N-sub})^+$ is the transitive closure of D_T/D_{N-sub}
N^{sup} is the superior characteristic graph at N

Fig. 8. Algorithm for creating the superior characteristic graph of a node in a tree

In Fig. 9 and Fig. 10 the incremental update algorithms are given. We start in the top node N of *newsub*. The attributes of N still have their original values. All other attributes of *newsub* have value ''unknown'' and index set to ''*TE*''. The index of all function attribute of N is set to ''*TE*'', the index of the parameter attributes of N, that have incoming edges in D_N^{sup} is also set to ''*TE*''. The index of all other attributes in the tree is ''*UC*''.

All functions on N that have ''*UC*'' parameter attributes can be evaluated: we use the values of the parameter attributes to construct a term $\phi(newsub, par(\phi,1), \ldots, par(\phi, n))$ and reduce this term using the algorithms of Fig. 6. Other attributes of *newsub* will get a value too, their indices then are set to ''*C*''. If the value of ϕ has not changed its index is set to ''*UC*'' and its successors in D_N^{sup} are checked. If a parameter attribute has only unchanged predecessors its own index is set to ''*UC*'' as well.

```
UPDATE–NEW–SUBTREE(Newsub, T)
N = top of Newsub
C–list := ∅
for all function attributes ϕ of N do
  index_ϕ := TE
  PROP–TobeEVAL–SUP(ϕ, N) od
Current Node := N
for Current Node do
  T' := tree rooted at Current Node
  for all attributes ϕ of Current Node which have
  index TE and parameters with index UC do
    construct term with ϕ, T' and parameters
    UPDATE(term, T') od
  if TE-attribute exists at Current Node then
  Current Node := parent of Current Node od
swap the index of all attributes of C–list to UC
create superior characteristic graphs for Newsub
END
```

T=attributed parse tree with an attribute dependency graph D_T,
Newsub is a subtree of T.
All nodes of T-Newsub have a superior characteristic graph
T' can be any subtree of T, T'_i is the i-th subtree of T'
C–list is a global variable for all changed attributes of T

```
UPDATE(term, T')
term is a closed Φ-term ϕ(p(x_1, ..., x_n), t_1, ..., t_m)
oldvalue := value_ϕ
term matches with the equation
ϕ(p(x_1, ..., x_n), y_1, ..., y_m) = τ
value_ϕ := UPDATE–FUNC(τ, eq_ϕp, T')
if oldvalue = value_ϕ then
  index_ϕ := UC
else
  index_ϕ := C, add ϕ to C–list
  PROP–TobeEVAL–UP(ϕ, T) fi
END
```

Fig. 9. Algorithms for updating attribute values after a subtree replacement

If a function attribute ϕ has obtained a new value, its successors in D_T must be re-evaluated. These successors are either function attributes of M, the parent node of N, or parameter attributes of N itself or parameter attributes of siblings of N. In either case, the new value of these successors can only be found by re-evaluating the (function) attribute of M that comes first when the dependency graph is followed, starting at the changed attribute. The algorithm *PROP–TobeEVAL–UP* of Fig. 11 searches this attribute using the following strategy. If a function attribute of N or one of its siblings gets index ''*TE*'', its successor attributes also get index ''*TE*''. If a parameter attribute gets index ''*TE*'', ''*TE*'' is propagated further via the function attributes belonging to this parameter. If a function attribute of M is reached, all its direct and transitive successors in M^{sup} get index ''*TE*'' as well. When no more functions of N can be evaluated, all functions of node M that have index ''*TE*'' and have unchanged parameters are re-evaluated. Those attributes of N that still need to be evaluated will get a new value when one of the ''*TE*'' functions of M is re-

```
UPDATE–FUNC (functerm, eq_φp, T')
functerm is a subterm of the right hand side of eq_φp
case functerm = parameter of φ with attribute a in T'
  return value_a
case functerm = f (s_1, . . . , s_k), f ∉ Φ
  for j = 1 to k do
    s_j:=UPDATE–FUNC(s_j, eq_φp, T') od
  return normal form of functerm
case functerm = ψ(x_i, v_1, . . . , v_k)
  if index_ψ= TE at T'_i then
    x_i:=T'_i
    for j = 1 to k do
      let b be attribute for par(ψ, j)
      if index_b= TE then
        old–value:=value_b
        value_b:=UPDATE–PAR(v_j, ψ, j, eq_φp, T')
        if old–value = value_b then
          index_b:= UC
        else
          index_b:= C, add b to C–list
          PROP–TobeEVAL–DOWN(b,T) fi fi
      v_j:=value_b od
    if all direct predecessors of ψ
    in D_T have index UC then
      index_ψ:= UC
    else
      old–value:=value_ψ
      functerm matches with equation
      eq_ψq with right hand side σ
      value_ψ:=UPDATE–FUNC(σ, eq_ψq, T'_i)
      if old–value = value_ψ then
        index_ψ:= UC
      else
        index_ψ:= C, add ψ to C–list fi fi fi
  return value_ψ
END
```

```
UPDATE–PAR(parterm, ξ, m, eq_φp, T')
parterm is (a subterm of) the term at the m-th parameter
position of ξ in the right hand side of eq_φp
case parterm = parameter of φ with attribute a in T'
  return value_a
case parterm = f (s_1,...,s_k), f ∉ Φ
  for j=1 to k do
    s_j:=UPDATE–PAR(s_j, ξ, m, eq_φp,T') od
  return normal form of parterm
case parterm = ψ(x_i, v_1, . . . , v_k)
  if index_ψ= TE at T'_i then
    x_i:=T'_i
    for j = 1 to k do
      let b be the attribute for par(ψ,j)
      if index_b= TE then
        old–value:=value_b
        value_b:=UPDATE–PAR(v_j, ψ, j, eq_φp, T')
        if old–value = value_b then
          index_b:= UC
        else
          index_b:= C, add b to C–list
          PROP–TobeEVAL–DOWN(b,T) fi fi
      v_j:=value_b od
    if all direct predecessors of ψ
    in D_T have index UC then
      index_ψ:= UC
    else
      old–value:=value_ψ
      parterm matches with equation
      eq_ψq with right hand side σ
      value_ψ:=UPDATE–FUNC(σ, eq_ψq, T'_i)
      if old–value=value_ψ then
        index_ψ:= UC
      else
        index_ψ:= C, add ψ to C–list fi fi fi
  return value_ψ
END
```

Fig. 10. Algorithms for re-evaluating function-terms and parameter-terms after a subtree replacement

evaluated. If the value of such a parameter attribute, say at node N', turns out to have been changed its direct and transitive successors in the nodes of the construction rooted at N' also get index "*TE*". The *PROP–TobeEVAL–DOWN* algorithm in Fig. 11 takes care of this. When no "*TE*"-functions in M are left the process stops. Otherwise, "*TE*"- functions on the parent of M are re-evaluated. After this process has stopped, all function attributes in T that are somehow connected to a function attribute of the top node have a correct value. Finally, all "*C*" indices are switched to "*UC*".

3.4. Comparison with the algorithm of Reps, Teitelbaum and Demers

The update algorithm described above is an adaptation of the *optimal time* algorithm of Reps, Teitelbaum and Demers [RTD83] for updating attributes after a subtree replacement. In their algorithm, as well as in ours, attributes are re-evaluated at most once and only if they have changed predecessors. Differences are caused by the fact that they have attribute rules for each attribute and because their algorithm applies for arbitrary non-circular attribute grammar. To point out similarities and differences we give a short description of the algorithm. The attributes dependencies that run from inherited attributes to synthesized attributes in a node are determined by the tree rooted at that node. A special graph, the *characteristic subordinate graph* N_{sub}, describes these dependencies. (We use the "parameter-of-function" relation instead). The update algorithm starts in the top N of *newsub* and with a "working-graph" called *Model*. Initially *Model* is $N^{sup} \cup N_{sub}$. All attributes of *Model* that have only direct predecessors with a known value are evaluated. To this end two list of attributes are kept. The list S for all attributes in *Model* that are ready to be evaluated because all their predecessors have got their final value, and the list *TobeEvaluated* for all attributes in *Model* with at least one predecessor that has obtained a new value. An attribute of S is

```
PROP–TobeEVAL–UP(a,T)
if a function attribute of N
  for all b such that (a,b)∈D do
    if Ind_b= UC then
      Ind_b:= TE
      if b parameter attribute of sibling of N
      or b parameter attribute of N then
        PROP–TobeEVAL–UP(b, T)
      else (b function attribute of parent of N)
        PROP–TobeEVAL–SUP(b, M)
      fi fi od
else (a parameter attribute of some node N)
  for all function attributes b of N
  such that a is parameter of b do
    if Ind_b= UC then
      Ind_b:= TE
      PROP–TobeEVAL–UP(b, T)
    fi od fi
END
```

```
PROP–TobeEVAL–SUP(a, M)
if a function attribute of M
  for all b such that (a,b)∈M^sup do
    if Ind_b= UC then
      Ind_b:= TE
      PROP–TobeEVAL–SUP(b, M)
    fi od
else (a parameter attribute of M)
  for all function attributes b
  such that a is a parameter of b do
    if Ind_b= UC then
      Ind_b:= TE
      PROP–TobeEVAL–SUP(b, M)
    fi od fi
END
```

```
PROP–TobeEVAL–DOWN(a,T)
let N be the node of T,
with a parameter attribute of N
for all b such that (a,b)∈D do
  if b parameter attribute of child of N then
    if Ind_b= UC then
      Ind_b:= TE
      PROP–TobeEVAL–HOR(b, T)
    fi fi od
END
```

```
PROP–TobeEVAL–HOR(a,T)
if a parameter attribute of node N
  for all function attributes b of N
  such that a is parameter of b do
    if Ind_b= UC then
      Ind_b:= TE
        PROP–TobeEVAL–HOR(b, T)
    fi od
else (a function attribute of some node N)
  for all b such that (a,b)∈D do
    if b parameter attribute of sibling of N
    or b parameter attribute of N then
      if Ind_b= UC then
        Ind_b:= TE
        PROP–TobeEVAL–HOR(b, T)
      fi fi od fi
END
```

Fig. 11. Algorithms for propagating "TE" in an attribute dependency graph

evaluated only if it is an element of *TobeEvaluated*. If a changed attribute of node N has a successor that is not in *Model*, *Model* is extended with one production, as follows: If the successor attribute is an attribute of the parent of N, N^{sup} is replaced in M by the dependencies of the upper production plus the subordinate graph of al the siblings of N plus the superior graph of the parent of N. If the successor is an attribute of one of the children of N, N_{sub} is replaced in M by dependencies of the production rooted at N plus the subordinate graphs of each of its children.

The top node and bottom nodes of *Model* are the same we reach when propagating *TE*. Our algorithms for propagating *TE* does not only select direct successors of changed attributes, but all attributes that lay either on the route to a function attribute of the parent node (*PROP–TobeEval–UP*), or on the route to a function attribute of a child node (*PROP–TobeEval–DOWN*). In a later stage the index of *TE* attributes may be set to *UC* when it is certain that none of its predecessors has obtained a new value. The set of attributes that get index *TE* sometime during the update process equals the set of attributes that become elements of S when the algorithm of Reps c.s. is applied. The set of attributes that are actually re-evaluated when applying our algorithm is the same as the set attributes that eventually become elements of *TobeEvaluated*.

Like the algorithm of Reps c.s, our update algorithm is an *optimal time* algorithm in the sense that the number of steps is proportional to the number of changed attributes. The number of successors or predecessors of an attribute is bound by the maximum number of attributes of an abstract tree constructor, that is the maximum number of functions (and parameters of these functions) defined on each sort of a constructor. The propagation of "*TE*" to the upper node is bound by this number and so is the propagation-down of "*TE*". This means that the number of all "*TE*" attributes is always proportional to the number of changed

attributes. "*TE*" attributes are either re-evaluated, which is counted as a unit step, or or its index is set to "*UC*", depending on the indices of its predecessors. The switching from "*C*" to "*UC*" in the end is obviously proportionally to the number of changed attributes.

In both update algorithms the optimal time property is lost when abstract tree constructors of variable arity, e.g. list constructors, occur in the grammar. Either because the number of steps of the propagate algorithm is no longer bound by a maximum or because the extension of *Model* is not bound to a limited number of nodes and attributes.

4. Conditional primitive recursive schemes

In this section we will describe how the incremental evaluation method of the previous section can be adapted to a *well-presented PRS with conditional equations*. In a PRS with conditional equations several defining equations for each pair (ϕ,p) may exist. It may happen as well that a certain Φ-term cannot be reduced. Conditional primitive recursive schemes are a superclass of primitive recursive schemes and provide more flexibility for writing specifications, especially for describing some aspects of dynamic semantics of languages.

We use $EQ_{\phi p}$ to indicate the list of equations with left-hand side $\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)$, $eq_{\phi pj}$ is the *j-th* equation in this list. The attribute dependencies for each abstract tree constructor are obtained by taking the union of the dependencies of all equations that apply to that constructor: $D_p=\cup_{\phi pj} D_{\phi pj}$.

In the non-conditional case an attribute dependency graph for an abstract syntax tree was constructed by simply patching the dependency graphs D_p of all occurrences of constructors p in the tree. But, if a certain term $\phi(p(T_1,\ldots,T_n),...)$ has to be reduced only one $eq_{\phi pj}$ will be used for this reduction. We do not want to have the dependencies of all other members of $EQ_{\phi p}$ in the attribute dependency graph of the tree $p(T_1,\ldots,T_n)$. Therefore, attribute dependencies for a tree are constructed upon reduction.

4.1. Restrictions on conditions

Conditions in Φ-equations may concern attributes, as well as subtrees. Consider the equation

$$\frac{\lambda_1=\rho_1\ ,\ \lambda_2=\rho_2}{\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau} \qquad \textbf{(eq2)}$$

If a condition $\lambda_k=\rho_k$ concerns an attribute, λ_k and ρ_k are of type $S\in S$, e.g. $f(y_j)=\cdots$ or $\psi(x_i,...)=\cdots$. If it concerns a subtree of $p(x_1,\ldots,x_n)$, λ_k is x_i and ρ_k is either x_j or some closed G-term. We reformulate the properties (v) to (vii) from Section 2.4 for a well-presented PRS with conditional equations.

(v′) All defining equations are strictly decreasing in G: in (eq2) the only terms of G that are allowed in τ *or in conditions concerning attributes* are the variables $x_1,\ldots,x_n$. If a condition concerns a G-term, λ_k equals x_i and ρ_k is either x_j or some closed G-term. Variables in conditions must have been introduced in the left-hand side of the equations (In Section 4.5 we will loosen this restriction). So, λ_k, ρ_k and τ are $S\cup\Phi\cup\{x_1,\ldots,x_n\}\cup\{y_1,\ldots,y_m\}$-terms.

(vi) For any two equations concerning the same function ϕ with conclusions

$$\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau\quad\phi(q(u_1,\ldots,u_{n'}),z_1,\ldots,z_m)=\tau',$$

the variable lists $<y_1,\ldots,y_m>$ and $<z_1,\ldots,z_m>$ are identical.

(vii′) In all occurrences of the form

$$\cdots\psi(x_i,v_1,\ldots,v_m)\cdots$$

in right-hand sides *or conditions* of defining equations of the same abstract tree constructor p, corresponding parameter terms v_j are identical.

(viii′) If two parameters $par(\psi,j)$ and $par(\xi,k)$ are represented by the same variable in left-hand sides of equations over the same constructor p, in all subterms

$$\psi(x_i,v_1,\ldots,v_m),\quad \xi(x_i,w_1,\ldots,w_{m'})$$

of right-hand sides *or conditions* of defining equations $eq_{\phi pj}$ and $eq_{\psi ph}$ v_j is identical to w_k.

In Example 4 and Example 5 the dynamic semantics of two different kinds of *if* statements are described. The evaluation functions *evs*, *eve*, *evss* are Φ-functions. In both examples the variable *Env* represents a value-environment: a table with identifiers and their values. The result of evaluating a statement is again a value-environment. In Example 4 the first argument of the *if* constructor is some integer expression which may have either value 0 or value 1. In Example 5 the first argument of the *if* constructor is either *true* or *false*.

Example 4

$$\frac{eve\,(Exp,Env) = 0}{evs(if(Exp, Stms_1, Stms_2), Env)=evss\,(Stms_1, Env)} \qquad \frac{eve\,(Exp,Env) = 1}{evs(if(Exp, Stms_1, Stms_2), Env)=evss\,(Stms_2, Env)}$$

Example 5

$$\frac{Bool = true}{evs(if(Bool, Stms_1, Stms_2), Env)=evss\,(Stms_1, Env)} \qquad \frac{Bool = false}{evs(if(Bool, Stms_1, Stms_2), Env)=evss\,(Stms_2, Env)}$$

4.2. Deducing attribute dependencies from a conditional equation

From a conditional equation $eq_{\phi pj}$ two sets of dependencies are deduced: $D\text{–}Cond_{\phi pj}$ from the conditions and $D\text{–}Rhs_{\phi pj}$ from the (right-hand side of) the conclusion, the union of the two is $D_{\phi pj}$. *D–Rhs* is similar to the dependencies deduced from non-conditional equations as described in Fig. 5. The deduction of *D–Cond* is described in Fig. 12. If a condition $\lambda_k{=}\rho_k$ concerns attributes only, dependencies are deduced from λ_k and ρ_k as if they were terms in the right-hand side of the conclusion. If a condition concerns a *G*-term x_i, an extra attribute, inc_i, is added to this term, and (inc_i, ϕ) is added to *D–Cond*, as obviously the value of ϕ somehow depends on the particular kind of subterm x_i. An *inc*-attribute does not have a value.

MAKEDEP–EQ$(eq_{\phi ph})$
$$eq_{\phi ph} = \frac{\lambda_1 = \rho_1\,, \ldots\,, \lambda_k = \rho_k}{\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau}$$
$D\text{–}cond_{\phi ph}$:=∅
for i=1 **to** k **do**
  MAKEDEP–COND$(\lambda_i, eq_{\phi ph})$
  MAKEDEP–COND$(\rho_i, eq_{\phi ph})$ **od**
$D\text{–}Rhs_{\phi ph}$:=∅
MAKEDEP–FUNC$(\tau, eq_{\phi ph})$
END

$D\text{–}Cond_{\phi ph}$ and $D\text{–}Rhs_{\phi ph}$ are global variables indicating the sets of dependencies of the conditions and the right hand side of the conclusion.

MAKEDEP–COND$(condterm, eq_{\phi ph})$
if $condterm{=}x_j$, $j \in \{1,\ldots,n\}$ **then**
  add(inc_j, ϕ) to $D\text{–}Cond_{\phi ph}$
else
  MAKEDEP–FUNC$(condterm, eq_{\phi ph})$ **fi**
END

Fig. 12. Algorithms for deducing attribute dependencies from a conditional equation

If an *inc*-attribute occurs in the attribute graph of an abstract syntax tree T, say inc_i in the top node of the subtree T_i, all nodes of T_i obtain an *inc*-attribute and the *inc*-attribute of each node in T_i depends on the *inc*-attribute of its children. The algorithm *EXTEND–INC*(D_T), not described here, arranges this in an obvious manner. Thus a re-evaluation of the ϕ-attribute that is the successor of inc_i is caused, after some change in T_i.

As will be described in the next section, the attribute dependency graph of each abstract syntax tree is always a subgraph of the pasting of all D_p of abstract tree constructors p that occur in the tree, extended with *inc*-attributes and *inc*-dependencies. If we ignore these *inc*-attributes, we can use the line of reasoning from Section 3.1 to show that each D_p is non-circular and that the pasting of these dependencies in a tree does not introduce any cycles either. Adding *inc*-attributes and *inc*-dependencies can not disturb the non-circularity as *inc*-attributes do not have incoming edges from other, i.e. non-*inc*, attributes.

In the construction of the superior characteristic graph of a node, *inc*-attributes are treated like function attributes: they may have outgoing edges to parameter attributes.

4.3. First time evaluation of a term

If we want to reduce a Φ-term $\phi(T, t_1, \ldots, t_n)$ using a conditional PRS, we start with a plain abstract syntax tree of T. Attributes and attribute dependencies are created dynamically: upon reduction of the Φ-term. Thus, we only get attribute dependencies that occur in the reduction. The advantage of such a dependency graph is that functions are not re-evaluated after an irrelevant subtree replacement. For instance, using the equations of Example 4, the evaluation of the term *evs*(*if*(*a*, *assign*(*c*,5), *assign*(*c*, 7)), *env*(*a*:0, *b*:1, *c*:3)) would result in a dependency graph as shown in Fig. 13.

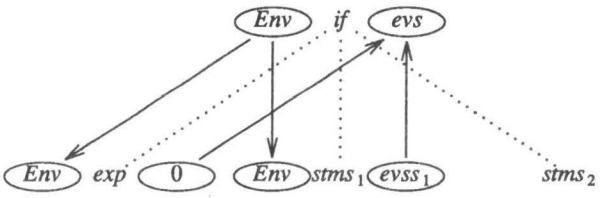

Fig. 13. Attribute dependencies for evaluation of an if-statement

Replacement of the subterm *assign*(*c*, 7) by some other statement would not cause a re-evaluation of *evs*(*if* (*a*, *assign*(*c*,5), *newterm*), *env*(*a*:0,*b*:1, *c*:3)) as the *evs* attribute has no incoming edges from any attribute of $stms_2$. We briefly describe the evaluation procedure as presented in the algorithms of Fig. 14 and Fig. 15.

REDUCE–NEW(*term*)
term is a closed Φ-term:
$\phi(p(T_1, \ldots, T_n), t_1, \ldots, t_m)$
let T be the term $p(T_1, \ldots, T_n)$, $D_T := \emptyset$
create attributes for ϕ and its parameters at T
for i=1 to m **do**
  t_i:= normal form of t_i
  value of $att(par(\phi, i)) := t_i$ **od**
SELECT–EQ(*term*,$EQ_{\phi p}$, D_T)=<*match–eq*, *D*>
$D_T := D$
if *match–eq*=$eq_{\phi ph}$ with right hand side τ **then**
  REDUCE–FUNC(τ, $eq_{\phi ph}$, T, D_T)=<*value*,*D*'>
  $value_\phi := value$
  $D_T := D'$
else
  $value_\phi$:=*term* **fi**
END

SELECT–EQ(*term*,$EQ_{\phi p}$, *D*)
match–eq:=*no–eq*, *j*:=1
while $j \le |EQ_{\phi p}|$ **and** *match–eq*=*no–eq* **do**
  REDUCE–COND($eq_{\phi pj}$, T, *D*)=<cond, *D*'>
  if cond = succeeded **then**
    match–eq:=$eq_{\phi pj}$
    D:=*D*'∪*D–Rhs*$_{\phi pj}$
  else if $j < |EQ_{\phi p}|$ **then**
      D:=*D*∪{(*att*(*par*(φ, *i*),φ) | 1≤*i*≤*arity*(φ)}
    else *j*:=*j*+1 **fi fi od**
return <*match–eq*, *D*>
END

REDUCE–COND($eq_{\phi ph}$, T, *D*)
let $eq_{\phi ph} = \dfrac{\lambda_1 = \rho_1, \ldots, \lambda_k = \rho_k}{\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau}$
D–temp:=*D*∪*D–Cond*$_{\phi ph}$
cond := succeeded
i := 0
while cond ≠ failed **and** *i*≤*k* **do**
  if $\lambda_i = x_j$ **then**
    $\lambda_i := T_j$
  else
    REDUCE–FUNC(λ_i, $eq_{\phi ph}$, T, *D–temp*)=<*value*,*D*>
    λ_i:=*value*
    D–temp:=*D* **fi**
  if $\rho_i = x_j$ **then**
    $\rho_i := T_j$
  else
    REDUCE–FUNC(ρ_i, $eq_{\phi ph}$, T, *D–temp*)=<*value*',*D*'>
    ρ_i:=*value*'
    D–temp:=*D*' **fi**
  if $\lambda_i = \rho_i$ **then**
    i := i + 1
  else
    cond := failed **fi od**
if cond = succeeded **then**
  D–temp:=*EXTEND–INC*(*D–temp*)
  return <*succeeded*, *D–temp*>
else
  return <*failed*,*D*>
END

Fig. 14. Algorithms for reducing a Φ-term using a conditional PRS

After the parameter terms $t_1, \ldots, t_n$ of $\phi(T, t_1, \ldots, t_n)$ have been normalized and stored in the appropriate attributes, the term matches the left-hand side of some equation. The conditions of this equation are checked one by one, during which a temporary dependency graph *D–Temp* is constructed. A condition succeeds if the normal form of the left-hand side equals the normal form of the right-hand side. If all conditions of an equation succeed the temporary dependencies are extended with *inc*-graphs and form the new D_T. Then the dependencies of the conclusion of the matching equation, *D–Rhs*, are added to D_T and the

right-hand side of the conclusion is reduced. If a condition fails, the temporary dependencies are removed and another equation is tried. If no matching equation is found whose conditions succeed, the reduction stops and the unreduced ϕ-term is stored in its attribute. Dependencies from all parameter attributes of ϕ to the ϕ attribute are added to D_T.

```
REDUCE-FUNC(functerm, eq_φρh, T', D)
functerm is a subterm of the right hand side of eq_φρh
or of a condition of eq_φρh
case functerm = parameter of φ with attribute a in T'
  return <value_a, D>
case functerm = f(s_1, ..., s_k), f ∉ Φ
  for j=1 to k do
    REDUCE-FUNC(s_j, eq_φρh, T',D)=<value, D'>
    s_j:=value, D:=D' od
  return <normal form of functerm,D>
case functerm = ψ(x_i, v_1, ..., v_k)
  if value_ψ = "unknown" at T'_i
    x_i:=T'_i
    for j=1 to k do
      let b be att(par(ψ,j)) at T'_i
      if value_b = "unknown"
        REDUCE-PAR(v_j, ψ, j, eq_φρh, T',D)=
                                  <value,D'>
        value_b:=value, D:=D' fi
      v_j:=value_b od
    SELECT-EQ(functerm, EQ_ψq, D)=<match-eq, D'>
    D:=D'
    if match-eq=eq_ψqh with right hand side τ then
      REDUCE-FUNC(τ, eq_ψqh, T'_i, D)=<value, D'>
      value_ψ:=value, D:=D'
    else (match-eq=no-eq)
      value_ψ:=functerm fi fi
  return <value_ψ, D>
END
```

```
REDUCE-PAR(parterm, ξ, m, eq_φρh, T', D)
parterm is (a subterm of) the term at the m-th parameter
position of ψ in the right hand side of eq_φρh
or in a condition of eq_φρh
case parterm = parameter of φ with attribute a in T'
  return <value_a, D>
case parterm = f(s_1, ..., s_k), f ∉ Φ
  for j=1 to k do
    REDUCE-PAR(s_j, ξ, m, eq_φρh, T', D)=<value, D'>
    s_j:=value, D:=D' od
  return < normal form of parterm,D >
case parterm = ψ(x_i, v_1, ..., v_k)
  if value_ψ = "unknown" at T'_i
    x_i:=T'_i
    for j=1 to k do
      let b be att(par(ψ, j)) at T'_i
      if value_b = "unknown"
        REDUCE-PAR(v_j, ψ, j,eq_φρh, T', D)=
                                  <value, D'>
        value_b:=value, D:=D' fi
      v_j:=value_b od
    SELECT-EQ(parterm, EQ_ψq, D)=<match-eq, D'>
    D:=D'
    if match-eq=eq_ψqh with right hand side τ then
      REDUCE-FUNC(τ, eq_ψqh, T'_i, newD)=<value,D'>
      value_ψ:=value
    else (match-eq=no-eq)
      value_ψ:=parterm fi fi
  return <value_ψ, D>
END
```

Fig. 15. Algorithms for reducing a function- and parameter-terms using a conditional PRS

4.4. Incremental evaluation with conditional equations

To illustrate the incremental construction and the updating of the dependency graph we return to the example in Fig. 13. When the expression *a* is changed into *b* the re-evaluation of the *eve* attributes yields a new value, 1, its successor, the *evs* attribute at the *if*-node, has to be re-evaluated. First, the incoming edges of the *evs* attribute are removed from D_T, then the term $evs(if(exp', stms_1 stms_2)env)$ is reduced using the second equation. During this reduction new dependencies are created for *evs* at the *if*-node, namely the dependencies of the second equation. This results in the dependency graph shown in Fig. 16.

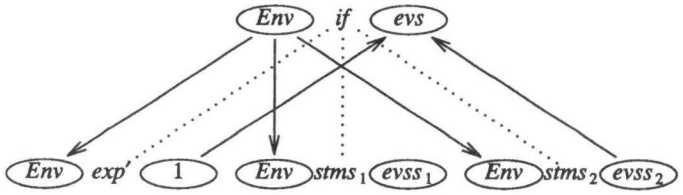

Fig. 16. Attribute dependencies after re-evaluation of an if-statement

The $evss_1$ attribute of $stms_1$ ($assign(c, 5)$) has been disconnected from its successor, so changes in this subterm will not cause a re-evaluation of the *evs*-attribute. Yet, the incoming edge to the parameter attribute *env* of $evss_1$ has not been removed, nor has any other attribute dependency in $stms_1$. Thus, it remains possible to communicate changes in the *env* attribute of the *if*-node to successor attributes of $stms_1$. If the value of the *env* attribute at the *if*-node changes, the index of its successors is set to "*TE*", and so is the index of all other attributes of $stms_1$ that depend on *env*. The effect of this "greedy" way of propagating

''*TE*'' is twofold. Firstly, after a subtree replacement in $stms_1$ re-evaluations of attributes that may have an incorrect value are prevented. When a subtree of $stms_1$ is replaced and all successors of the *env*-attribute at the top of $stms_1$ have index ''*TE*'', none of these attributes will be re-evaluated because the *UPDATE(Newsub,T)* algorithm takes care only of re-evaluating all ''*TE*'' function attributes of which the parameter attributes have index ''*UC*''. In the non-conditional case all attributes were somehow connected to a function attribute of the top node, and therefore would be re-evaluated eventually. When the $evss_1$-attribute at the top of $stms_1$ has been disconnected from its successor it will not be re-evaluated, nor will its parameter attribute. Secondly, when the value of the *eve* attribute at *exp* becomes 0 again the connection between $evss_1$ attribute and the *evs* attribute will be restored. If the $evss_1$-attribute at $stms_1$ has index ''*UC*'' at that moment, we can savely reuse its value to find the value of the *evs*-attribute. If the index of $evss_1$-attribute is ''*TE*'', it will be re-evaluated meanwhile also re-evaluating all its ''*TE*'' predecessors in $stms_1$. After a re-evaluation all attributes that played a role in this re-evaluation have a correct value, other attributes either have a correct value or have index ''*TE*''.

The algorithm used for incremental evaluation with a dynamically created dependency graph is very similar to the update algorithms of Fig. 9 and Fig. 10. Adaptations are:

- Before a function attribute is re-evaluated its incoming edges are removed from D_T.
- For re-evaluation of a function attribute the algorithm *SELECT–EQ* of Fig. 14 is used to find a matching equation. The dependencies of this equation are used to create new incoming edges for the function attribute.
- The *PROP–TobeEVAL–DOWN* is replaced by a more greedy one. The indices of all direct and transitive successors of a changed parameter attribute in an underlying subtree are set to ''*TE*''.
- When a subtree is replaced, the index of its *inc* attribute is set to ''*C*''. *PROP–TobeEVAL–UP* and *PROP–TobeEVAL–SUP* are adapted so that the indices of *inc* attributes with changed or ''*TE*'' predecessor are set to ''*C*'' as well.
- The check on ''*TE*''-attributes of the *Current–Node* in *UPDATE(Newsub,T)* must be extended with a check on the existence of an *inc*-attribute with index ''*C*''. In both cases attributes of the parent node must be re-evaluated.

The adapted algorithm is not optimal time: The number of steps in the greedy *PROP–TobeEVAL–DOWN* algorithm does not depend on the number of affected attributes nor is it bound by a constant, it is only bound by the number of attributes in a tree. However, simply switching an index is not a costly operation.

4.5. Allowing new variables in conditions.

The term rewriting engine in the ASF+SDF-system can handle a limited class of conditional equations [Hen88]. The limitations concern the use of ''new'' variables in conditions, that is variables that have not been introduced in the left hand side of the conclusion of the equation. New variables may be introduced in one side of each condition, moreover the term in which a new variable occurs must be in normal from. When the condition is checked all known variables are instantiated, hence a closed term and an open term are obtained. The condition succeeds if the normal form of the closed term matches the (unreduced) open term at the other side of the condition. Then an instantiation of the new variable is found and the new variable is added to the list of known variables that can be used for the instantiation of the terms in subsequent conditions and in the right-hand side of the equation. If we would allow introduction of new variables within these limitations no problems would arise for conditions concerning attributes, e.g. the following two equations would give the same dependency graphs

$$\frac{\psi(x_1) = newvar,\ \chi(x_2){=}h(newvar)}{\phi(p(x_1, x_2), y) = true} \qquad \frac{\chi(x_2) = h(\psi(x_1))}{\phi(p(x_1, x_2), y) = true}$$

The algorithm for deducing a dependency graph from an equation only needs little adaptation to handle these conditions. But when new variables concerning a *G*-term are introduced in a condition, dependencies may be introduced between attributes of nodes that do not belong to one constructor. e.g:

$$\frac{x_1 = q(newvar\,1,\ newvar\,2))}{\phi(p(x_1, x_2), y) = \psi(newvar\,2,\ y)}$$

would cause a dependency between the attributes at the top node of *newvar*2 and the attributes at the "grandparent" node *p*. Generally spoken, by allowing new variables concerning *G*-terms in conditions dependencies may be constructed between attributes of any pair of nodes in a tree. On the one hand, allowing the use of new variables of a sort of *G* in conditions would considerably complicate our algorithm. On the other hand it is not very likely to occur in language definitions and can easily be avoided by adding an extra function for the sort of X_1, so for the moment we have chosen not to adapt our algorithm to this situation.

4.6. Relaxing the disjointness requirement

So far, we assumed that the sorts of *S* and *G* in a PRS <*G*,*S*, Φ> were disjoint. Yet, in existing typecheck specifications many sorts used in the specification of the syntax of a language are also used in the description of its semantics, especially trivial sorts like Booleans, natural numbers and identifiers. For instance, in the ASPLE typecheck specification a declaration, consisting of a type followed by a list of identifiers, brings forth a table (type-environment) of identifiers together with their types. Hence, the sorts Identifiers and Types of the syntax of ASPLE are needed to describe these type-environments. Equations (**eq5**) and (**eq6**) are the common way to define typechecking of a declaration.

$$tcdecl(decl(Type, Idlist), Env) = tclist(Idlist, Type, Env) \qquad \textbf{(eq5)}$$

$$tclist(list(Id, Idlist), Type, Env) = tclist(Idlist, Type, add(pair(Id, Type), Env)) \qquad \textbf{(eq6)}$$

The problem arises that the argument *Type* in the right hand sides of these equations refers to a subtree instead of a function attribute, so does the argument *Id* in the right hand side of (**eq6**). The trick with the *inc* attribute solves the problem as is shown in Fig. 17.

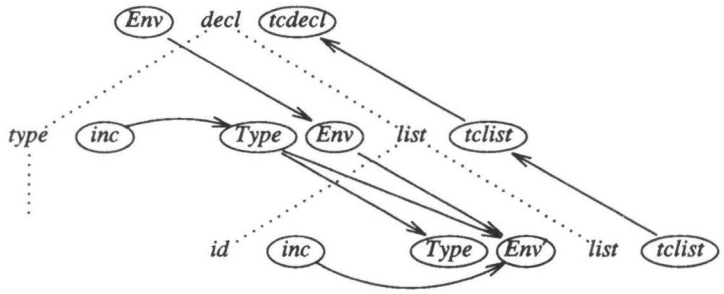

Fig. 17. Attributed subtree for typechecking declarations in ASPLE

When the disjointness requirement is relaxed, the elements of Φ should be indicated explicitly, and the strictly decreasing property as formulated in (v) and (v′) should be rephrased as

(v″) the first argument of a Φ-term in the right hand side of an equation or in conditions of a defining equation should be a direct subterm of the first argument of the left hand side of that equation.

Note, that we still hold on to the requirement that no equations exist over sorts of *G*.

Now, the specifications of the typechecking of ASPLE and mini-ML as described in [Meu88, Hen89, Hen91] fall within the class of specifications for which an incremental implementation can be made. The specification of the static semantics of Pascal as described in [Deu91] needs some minor changes to make it fit into this class.

5. Related Work

As far as we know, the only paper in which some kind of incremental term rewriting is mentioned is [FGJM85]. In that paper the term rewriting engine of OBJ2 is briefly described. OBJ2 is a functional programming language based upon equational logic and implemented as a term rewriting system. The computation of terms with a top symbol that has the "saveruns" annotation (set by the user) are stored in a hash-

table. Before a term is computed this table is checked.

There are many papers dealing with the relation between attribute grammars and other formalisms. Most of these have different purposes from ours, however, like solving the space problem attribute grammars usually cause. The results of Courcelle and Franchi-Zannettacci are used in papers by Attali and Franchi-Zannettacci and Jourdan [AF88, Att88, Jou84]

Jourdan starts with a strongly non-circular attribute grammar and compiles it to a primitive recursive scheme. He does not store inherited attributes thus trading time for space.

Attali and Franchi-Zannettacci translate TYPOL programs to attribute grammars. TYPOL is a formalism for specifying the semantics of programming languages [DT89, CDDHK85] and is closely related to PROLOG. Its original implementation has accordingly been in that language. In order to avoid the unification of PROLOG and make incremental and partial evaluation of TYPOL programs possible, TYPOL programs that are pseudo-circular and strictly decreasing are completely translated to (strongly non-circular) attribute grammars, in either the Synthesizer Specification Language, SSL [RT89b, RT89a] or Olga, the input formalism for the FNC-2 system [JLP90, JP88].

Katayama [Kat84] describes how strongly non-circular attribute grammars can be translated to procedures, considering non-terminals as functions that map inherited attributes to synthesized attributes. He extends his method to general non-circular attribute grammars.

6. Conclusion and further research.

We have described how incremental evaluation of terms can be derived from well-presented primitive recursive schemes and from the larger class of conditional well-presented primitive recursive schemes. The derivation is based on the isomorphism between well-presented primitive recursive schemes and strongly non-circular attribute grammars.

The class of well-presented primitive recursive schemes is a very natural one for specifying the static semantics of languages. The class of conditional well-presented primitive recursive schemes provides more flexibility and many aspects of the dynamic semantics of languages can be described in this context. Further research will concern extending our method to a superclass of primitive recursive schemes in which also the evaluation of the while statement can be described, e.g.

$$\frac{eve(Exp, Env) = true}{evs(while(Exp, Stms_1, Stms_2), Env) = evs(while(Exp, Stms_1, Stms_2), evs(Stms_1, Env))}$$

This means an extension to conditional primitive recursive schemes that are not necessarily well-presented, possibly cyclic, and decreasing instead of strictly decreasing.

The incremental evaluation method described in this paper will be implemented in the term rewriting system for algebraic specifications of the ASF+SDF-system. In order to be able to mix incremental evaluation of terms with normal evaluation the incremental strategy is an adaptation of the leftmost innermost reduction strategy used in this system. The implementation will be written in LeLisp [LeLisp87].

Acknowledgements

I would like to thank Jan Heering and Paul Klint for many discussions on incremental evaluation. Arie van Deursen, Jan Heering, Paul Hendriks and Paul Klint made helpful comments on previous versions of this paper.

References

[Att88] I. Attali, "Compiling TYPOL with attribute grammars," in: *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming '88*, ed. P. Deransart, B. Lorho, and J. Małuszyński, Lecture Notes in Computer Science **348**, Springer-Verlag, pp. 252-272 (1988).

[AF88] I. Attali and P. Franchi-Zannettacci, "Unification-free execution of TYPOL programs by semantic attribute evaluation," in: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, ed. R. Kowalski and K. Bowen, Logic Programming

Series, MIT Press, pp. 160-177 (1988).

[JLP90] M. Jourdan, C. Le Bellec, and D. Parigot, "The Olga attribute grammar description language: design, implementation and evaluation," in: *Attribute grammars and their applications - Proceedings of the WAGA conference*, ed. P.Deransart M. Jourdan , Lecture Notes in Computer Science **461**, Springer-Verlag, pp. 222-237 (1990).

[BHK89] J.A. Bergstra, J. Heering, and P. Klint (eds.), *Algebraic Specification*, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley (1989).

[CDDHK85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn, "Natural semantics on the computer," Rapports de Recherche 416, INRIA, Sophia Antipolis (1985).

[CF82a] B. Courcelle and P. Franchi-Zannettacci, "Attribute grammars and recursive program schemes I," *Theoretical Computer Science* **17**, pp. 163-191 (1982).

[CF82b] B. Courcelle and P. Franchi-Zannettacci, "Attribute grammars and recursive program schemes II," *Theoretical Computer Science* **17**, pp. 235-257 (1982).

[DJL88] P. Deransart, M. Jourdan, and B. Lorho, *Attribute Grammars - Definitions, Systems and Bibliography*, Lecture Notes in Computer Science **323** , Springer-Verlag (1988).

[DT89] T. Despeyroux and L. Théry, *TYPOL - User's guide and manual*, The CENTAUR Documentation - Version 0.9, Volume I - User's Guide, INRIA, Sophia-Antipolis (1989).

[Deu91] A. van Deursen, "An algebraic specification for the static semantics of Pascal," Report CS-R91??, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1991), to appear.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," in: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ed. B. Reid, ACM, pp. 52-66 (1985).

[Hen88] P.R.H. Hendriks, "ASF system user's guide," Report CS-R8823, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988), Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands, CSN'88* **1**, pp. 83-94, SION (1988).

[Hen89] P.R.H. Hendriks, "Typechecking Mini-ML," in: *Algebraic Specification*, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 299-337 (1989), Chapter 7.

[Hen91] P.R.H. Hendriks, *Implementation of Modular Algebraic Specifications*, University of Amsterdam (1991), Chapter 4, to appear.

[Jou84] M. Jourdan, "Strongly non-circular attribute grammars and their recursive evaluation," in: *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, ACM, pp. 81-93 (1984), Appeared as SIGPLAN Notices 19(6).

[JP88] M. Jourdan and D. Parigot, "The FNC-2 system: advances in attribute grammar technology," Rapports de Recherche 834, INRIA, Rocquencourt (1988).

[Kas84] U. Kastens, "The GAG-System - A tool for compiler construction," in: *Methods and Tools for Compiler Construction*, ed. B. Lorho, Cambridge University Press, pp. 165-181 (1984).

[Kat84] T. Katayama, "Translation of attribute grammars into procedures," *ACM Transactions on Programming Languages and Systems* **6**(3), pp. 345-369 (1984).

[KP90] K. Koskimies and J. Paakki, *Automating Language Implementation* , Ellis Horwood (1990).

[LeLisp87] *LeLisp, Version 15.21, le manuel de référence*, INRIA, Rocquencourt (1987).

[Meu88] E.A. van der Meulen, "Algebraic specification of a compiler for a language with pointers," Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988).

[RT89a] T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual - Third edition*, Springer-Verlag (1989).

[RT89b] T. Reps and T. Teitelbaum, *The Synthesizer Generator: a System for Constructing Language-Based Editors*, Springer-Verlag (1989).

[RTD83] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Transactions on Programming Languages and Systems* **5**(3), pp. 449-477 (1983).