1991

E. A. van der Meulen

Fine-grain incremental implementation of algebraic specifications

Computer Science/Department of Software Technology Report CS-R9159 December

Fine-Grain Incremental Implementation of Algebraic Specifications

E.A.van der Meulen

CWI, P.O.Box 4079, 1009 AB Amsterdam. Email: emma@cwi.nl

Abstract

Considering symbol tables as aggregate values often forms an obstacle to incremental type checking or evaluation of programs. A small change in a symbol table requires many expensive re-calculations of values that depend on (part of) the symbol table.

In a previous paper we described how an incremental implementation can be derived from algebraic specifications belonging to the subclass of well-presented primitive recursive schemes. Type check specifications typically belong to this class. In this paper we generalize this incremental implementation method to functions on values of auxiliary data types like symbol tables. Thus we obtain fine-grain incremental implementations. This fine-grain incrementality can be derived from a subclass of algebraic specifications that we will call layered primitive recursive schemes.

Key Words & Phrases: Software Engineering, Algebraic Specification, Incremental Evaluation, Attribute Grammars, Higher-Order Attribute grammars, Attribute Dependency Graphs.

1991 Mathematics Subject Classification: 68N20 **[Software]:** Compilers and generators; 68Q42 **[Theory of computing]:** Rewriting systems; 68Q50 **[Theory of computing]:** Grammars; 68Q65 **[Theory of computing]:** Abstract data types, algebraic specification.

1991 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 [Meu90] we described how an incremental implementation can be derived from algebraic specifications belonging to the subclass of conditional well-presented Primitive Recursive Schemes with parameters (PRS for short). The implementation is based on the equivalence between such PRS's and strongly non-circular attribute grammars [CFZ82]. For each so-called *incremental* function attributes are associated with the sort of its first argument: an inherited attribute for each of the other arguments plus one synthesized attribute for the result of the function.

For example, the type checking of a program and its substructures can often be implemented by means of incremental functions. The program is stored as an attributed term.

Report CS-R9159
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Editing the program corresponds to a subterm replacement and is followed by an attribute update computation to obtain the new type check value.

A first experimental implementation of our method exists, which not only shows that incremental evaluation is often advantageous, but also that it is sometimes much slower than completely re-evaluating the modified term. Attribute updating can be very expensive if a changed attribute has many successors that all obtain a new value as well and that are used in expensive calculations. This is, for example, the case if the result of type checking declarations yields a table (so the corresponding attribute contains an *aggregate* value), which is used to type check the variables in the statement section of the program. A small change in this table causes an attribute update computation that affects the complete statement section. In particular, for each variable a lookup operation on the complete modified table has to be performed to find its type.

We propose a refinement of our existing incremental technique for operations on programs and its substructures, to operations on attributes. The resulting fine-grain incrementality allows incremental evaluation of attributes by storing *auxiliary attributed terms* for them. A change in a predecessor attribute then causes the attributes of this auxiliary term to be updated. Furthermore, we define the class of *layered primitive recursive schemes*, for which a fine-grain implementation can be derived. Finally, we improve our technique on time and space use by making use of the fact that very often auxiliary terms are all copies of a single attribute. In that case one *multiply attributed* copy is stored together with a hash table.

Whereas the method we describe is general in the sense that it does not need predefined data types, but can be applied to any layered Primitive Recursive Scheme, the specification of a *lookup* function on a table data type in a type check specification serves as a running example and will be its main field of application. Moreover, the proposed technique can be seen as a first step towards an incremental implementation of construction, updating and lookup operations on symbol tables for different languages with a variety of scope rules, without changing or extending the specification formalism.

1.1 Related work

As related work we consider methods for handling symbol tables in the context of other specification formalisms like attribute grammars and higher order attribute grammars [HT86a, HT86b, VSK90], a model for incremental symbol processing [Fri88], and the incremental language designed by Yellin and Strom [YS91].

The principle of applying incremental methods for operations on programs and their substructures to operations on attributes has been used earlier by Vogt et.al. [VSK90] in implementing higher order attribute grammars. They generalize their attribute evaluation technique for ordered attribute grammars, based on visit sequences, to one for ordered higher order attribute grammars. When a symbol table is described by means of grammar rules with attributes for lookup, their approach can be used, like ours, to obtain an incremental implementation of operations on symbol tables. In Section 5 we will further investigate the relationship between layered primitive recursive schemes and higher order attribute grammars.

In [HT86a] Hoover and Teitelbaum describe how symbol tables, can be dealt with efficiently in the Synthesizer Generator [RT89], an attribute grammar based system for specifying languages. A special data-type, called finite function data type, is added to the attribute grammar specification language. This finite function data type can be used to represent symbol tables. Predefined operations on the data type for construction, updating and lookup have an incremental implementation.

Another approach to make attribute grammars more efficient is found in [HT86b]. Horwitz and Teitelbaum describe relationally attributed grammars. An attribute grammar is augmented with a relational database. Attribute values can be used to construct relations and values from relations can serve as input to attribute equations. Views on relations are updated incrementally. Among other things, a symbol table can be maintained as a relation, with views defined to find the types of variables. Changing a variable declaration then causes a change in the symbol table which triggers an incremental update of its views.

In his design of a multi-purpose incremental symbol processing system, Fritzson [Fri88], uses an entity-relational model, a model which combines the relational approach with the object oriented approach. An incremental implementation exists for insertion and deletion of relations, whereas direct access to the symbol table makes the lookup operation efficient. The system can be coupled to an incremental compiler or an interactive editing environment through its lookup and define operations.

INC, [YS91], is a language for incremental computation in general. Apart from elementary data-types it supports the constructed data-types tuple and bag. An incremental implementation exists for operations on these data-types. A program in INC is implemented as a network of processes which transfer data. When changes in input data of a process are given as messages communicating the difference with the previous input data, the process, in turn, generates output messages describing the difference with its previous output. Although INC is not a language specification formalism, an example is given of a type checker in INC. The program consumes bags of declarations, scope information, and references and produces a bag of resolved variables (with their types) and another bag of unresolved variables. The program could be coupled to an interactive editing environment.

1.2 Organization of this paper

In Section 2 we briefly explain primitive recursive schemes and their incremental implementation. We illustrate the shortcoming of the implementation obtained in this way and give a sketch of the solution. Section 3 gives a definition of layered primitive recursive schemes and describes how fine-grain incremental evaluators can be derived from specifications belonging to this class. In Section 4 is described how fine-grain incrementality can be improved by taking copy dependencies into account is described. Section 5 discusses the relation between layered primitive recursive schemes and higher-order attribute grammars. Finally, conclusions and future plans are presented in Section 6.

2 Primitive recursive schemes

A conditional well-presented primitive recursive scheme with parameters (PRS for short) is an algebraic specification that can be described by a 5-tuple $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$. The signature of the specification is formed by the union of G, S and Φ, and the equations of the specification are Eq and Eq_Φ.

All functions in G are free constructors, that is, no equations over G-terms exist in $Eq \cup Eq_\Phi$. For each function $\phi \in \Phi$ the type of its first argument is a sort of G, and the types of the other arguments, called the *parameters* of ϕ are sorts of S, and so is the output sort. Intuitively, G represents the abstract syntax of a language, Φ the type check constructors:

program	:	DECLS	× STMS	→	PROGRAM
empty-decls	:			→	DECLS
decls	:	DECL	× DECLS	→	DECLS
stms	:	STM	× STMS	→	STMS
assign	:	ID	× EXP	→	STM

Φ-functions:

tcp	:	PROGRAM		→	BOOL	{incremental}
tcdecls	:	DECLS	× TENV	→	TENV	{incremental}
tcdecl	:	DECL	× TENV	→	TENV	{incremental}
tcstms	:	STMS	× TENV	→	BOOL	{incremental}
tcstm	:	STM	× TENV	→	BOOL	{incremental}
tcexp	:	EXP	× TENV	→	TYPE	{incremental}

Φ-defining equations:

[Tc1] *tcp(progam*(Decls,Stms)) = *tcstms*(Stms,*tcdecls*(Decls, *empty-tenv*))
[Tc2] *tcdecls(empty-decls*, Tenv) = Tenv
[Tc3] *tcdecls(decls*(Decl, Decls), Tenv) = *tcdecls*(Decls,*tcdecl*(Decl, Tenv))
[Tc4] *tcstms(stms*(Stm,Stms),Tenv) = *and(tcstm*(Stm,Tenv),*(tcstms*(Stms,Tenv)))
[Tc5] *tcstm(assign*(Id,Exp),Tenv) = *compatible(lookup*(Tenv,Id),*tcexp*(Exp,Tenv))

Figure 1: Part of an algebraic specification of a type checker

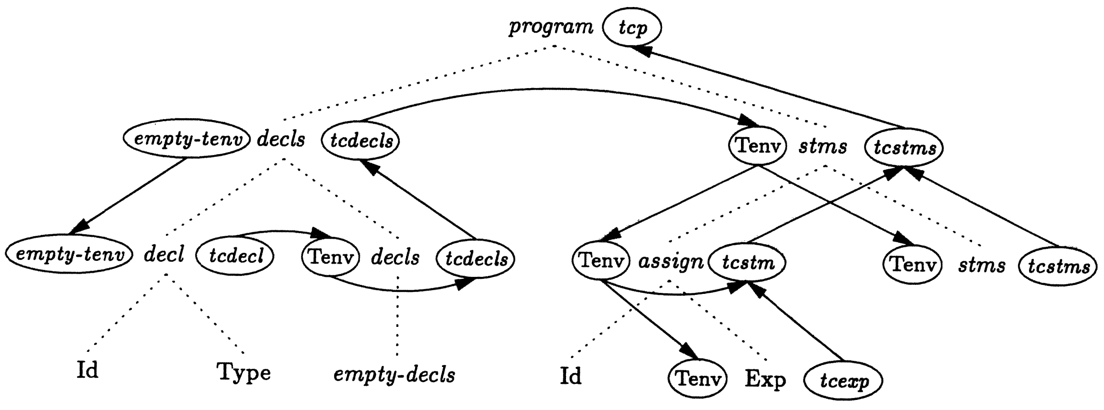


Figure 2: Top of an attributed term

and evaluation functions, and $S \cup Eq$ the specification of the auxiliary data types, used for describing type checking and evaluation. G and S need not be disjoint.

Eq_Φ is the set of so-called Φ-defining equations. For each pair (ϕ, p), with p a constructor of G to which ϕ applies, Eq_Φ contains an equation. A defining equation has the form

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{1}$$

All x_i and y_j in the left-hand side are different variables. If Φ-terms occur in the right-hand side τ their first argument is some x_i, $i \in 1, \ldots, n$. It is also possible that defining equations have conditions $\lambda_1 = \rho_1, \ldots, \lambda_k = \rho_k$. In that case several defining equations may exist for one pair (ϕ, p). Since terms λ_i and ρ_i must meet the same requirements as the term τ at the right-hand side of the conclusion and are processed in the same way we will consider equation 1 the general form.

We refer to Appendix A for a precise account of the properties of well-presented primitive recursive schemes. Here we illustrate the notion with an example. Figure 1 presents part of an algebraic specification of the type checker of a simple programming language. This specification is a PRS, with *program, decls, stats, assign* the constructors of the abstract syntax of the language, and the type check functions, *tcp, tcdecls, tcdecl, tcstms* and *tcstm* the Φ-functions or *incremental* functions. We can tell from their signature that type checking a program yields a Boolean value. When declarations are type checked a type environment is constructed, that is, a table with identifiers and their types. The signature of the type environment is not described here but will be presented shortly. A type environment is used as a second argument for type checking both declarations and statements. Hence, we call TENV a *parameter* of the functions *tcdecls, tcdecl, tcstms* and *tcstm.* The specification of the Booleans and the type environments typically forms the specification of the auxiliary data types: $S \cup Eq$.

The equations describe how a program and its substructures are type checked. Equation [Tc1] is the defining equation for the pair (*tcp,program*), [Tc2] and [Tc3] are defining equations for (*tcdecls,empty-decls*) and (*tcdecls,decls*) respectively, while [Tc4] and [Tc5] are defining equations for (*tcstms,stms*) and (*tcstm,assign*).

2.1 Incremental evaluation

On the one hand, PRS's form a subclass of algebraic specifications, on the other hand they can be considered as a way of describing attribute grammars [CFZ82, DJL88]. This allows us to transfer techniques for incremental evaluation of attributes in a dependency graph to incremental evaluation of terms in an algebraic specification.

In a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$we associate attributes with the sorts of the abstract syntax G. Each function $\phi \in \Phi$ corresponds to attributes of the sort of its first argument: one synthesized attribute and an inherited attribute for each parameter. *Attribute dependencies* for a constructor can be derived from the defining equations. From equation 1 dependencies are derived as follows.

1. To determine the predecessor attributes for the synthesized attribute associated with ϕ at the top node of $p(x_1, \ldots, x_n)$, take all Φ-terms and y_j's in τ that are not subterms of a Φ-term. The attributes associated with these terms are predecessors of ϕ.

2. To determine the predecessor attributes for an inherited attribute at a sub-node of $p(x_1, \ldots, x_n)$ and associated with a parameter term in τ, take all Φ-terms and y_j's in

constructors:

pair	: ID	× TYPE	→ PAIR	
empty-tenv	:		→ TENV	
tenv	: PAIR	× TENV	→ TENV	

Φ-functions:

lookup	: TENV	× ID	→ TYPE	{incremental}
id-of	: PAIR		→ ID	{incremental}
type-of	: PAIR		→ TYPE	{incremental}

Φ-defining equations:

[Tenv1] $$\frac{\textit{id-of}(\text{Pair}) = \text{Id}}{\textit{lookup}(\textit{tenv}(\text{Pair},\text{Pairs}),\text{Id}) = \textit{type-of}(\text{Pair})}$$

[Tenv2] $$\frac{\textit{id-of}(\text{Pair}) \neq \text{Id}}{\textit{lookup}(\textit{tenv}(\text{Pair},\text{Pairs}),\text{Id}) = \textit{lookup}(\textit{tenv}(\text{Pairs}),\text{Id})}$$

[Tenv3] *lookup(empty-tenv*,Id) = *error-type*
[Tenv4] *id-of(pair*(Id,Type)) = Id
[Tenv5] *type-of(pair*(Id,Type)) = Type

Figure 3: Part of an algebraic specification of a type environment

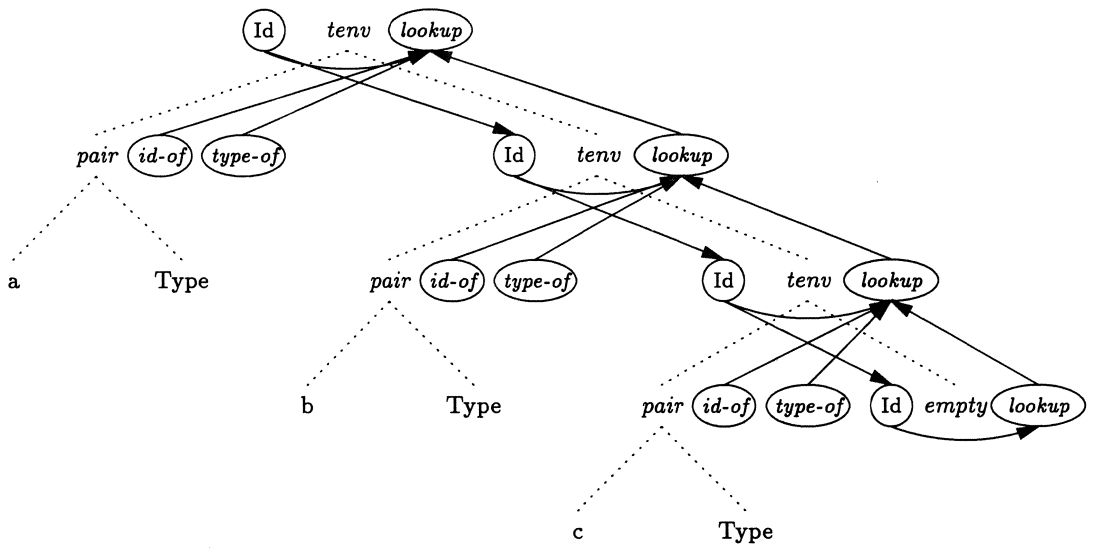


Figure 4: An attributed type environment

the parameter term that are not subterms of a Φ-term. The attributes associated with these terms are predecessors of the inherited attribute.

When a term *tcp*(Program) has to be reduced the term Program is stored and decorated with attributes which are connected by means of a dependency graph, as shown in Figure 2. During reduction, rewriting of a Φ-term or parameter term is preceded by checks on the corresponding attributes of Program. If the attribute does not contain a value, the term is reduced and its normal form is stored in the attribute. If the attribute does contain a value, reduction can be skipped for this value is the normal form of the term to be reduced.

Editing Program corresponds to replacing a subterm in the stored term and triggers an attribute updating process starting at the replacement node. An attribute is re-evaluated if and only if one of its predecessors has changed value.

2.2 Shortcoming of incremental evaluation

The shortcoming of our incremental algorithm can easily be illustrated. A small modification in the declaration section causing an equally small modification in the top *tcdecls* attribute, requires not only a, possibly extensive, reconstruction of the top *tcdecls* attribute, but also a re-evaluation of the complete statement section. Whereas much of this re-evaluation is simply copying new values, a term *lookup*(NewTenv,Id) has to be reduced for every identifier occurrence in the statements. This can be an expensive operation even when the modification does not affect (the type of) the identifier looked for. We will show how an incremental implementation could solve this problem. We will not deal here with the improvements for the reconstruction of the *tcdecls* attribute.

2.3 Incremental evaluation of the *lookup*

In Figure 3 part of an algebraic specification of a type environment is presented. Note that this algebraic specification is a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$. G is formed by the sorts ID, TYPE, PAIR and the constructors *pair, empty-tenv* and *tenv.* The constructors for TYPE are not shown. S contains the sorts ID and TYPE as well. *lookup*, *id-of* and *type-of* are the elements of Φ and equations [Tenv1]-[Tenv5] are the elements of Eq_Φ. Eq is empty. Instead of completely re-evaluating the term *lookup*(Tenv,Id) after each modification in the type environment, we could store a copy of the type environment and decorate it with *Id* and *lookup* attributes, as shown in Figure 4. The normal form of the *lookup*-term can be determined incrementally, by attribute updating. If the modification in the type environment does not affect the identifier that is checked, re-evaluation will be fast.

3 Layered PRS's and fine-grain incremental evaluation

In this section, we first define the class of layered primitive recursive schemes which is needed for fine-grain incremental evaluation of algebraic specifications. Next, we explain how so-called auxiliary attributed terms can be derived from a layered PRS and how they are used to improve the incremental implementation. We describe how attribute dependencies are derived from equations and conclude with a description of the attribute update procedure.

3.1 Layered primitive recursive schemes

Definition. An algebraic specification is a *layered PRS* if it is a PRS, $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ for which holds that

(ix) The algebraic specification $G_1 \cup S_1 \cup Eq_1$ is a PRS as well, say $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$.

(x) Moreover, if in a Φ_1-defining equation

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$$

a Φ_2-term occurs in the right-hand side τ or in a conditional term λ_j or ρ_j, its first subterm is *not* x_i, $i \in 1, \ldots, n$.

We call $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ a *sub-PRS* of $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$. A sub-PRS can be layered itself. Property (x) guarantees a useful subdivision of a specification into PRS's. Without this property one could divide the PRS of Figure 1 in several ways into a layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ with sub-PRS $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$. For instance with *tcp, tcdecls, tcdecl* in Φ_1 and *tcstms, tcstm* elements of Φ_2.

A PRS is *well-presented* if the parameter terms for all functions in Φ are uniquely defined for each constructor they apply to (see appendix A). A layered PRS is *well-presented* if both $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ and $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ are well-presented. The specification of the type checker in Figure 1 together with the specification of the type environment in Figure 3 is a well-presented layered PRS. Note that in Φ_1-defining equations, well-presentedness does not affect the Φ_2-subterms. For instance, in the equation

$$\mathit{tcexp}(\mathrm{Id}_1 + \mathrm{Id}_2, \mathrm{Tenv}) = \mathit{compatible}(\mathit{lookup}(\mathrm{Tenv}, \mathrm{Id}_1), \mathit{lookup}(\mathrm{Tenv}, \mathrm{Id}_2)) \tag{2}$$

the two *lookup* terms in the right-hand side have different parameters. This is allowed in a well-presented layered $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ in which *tcexp* is an element of Φ_1 and *lookup* an element of the incremental functions of its sub-PRS $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$.

As we are going to use Φ_2-terms for the incremental computation of Φ_1-terms, we are interested in those layered PRS's in which Φ_2-terms do occur in the reduction of Φ_1-terms. Therefore, we introduce the notion of *nested* sub-PRS's.

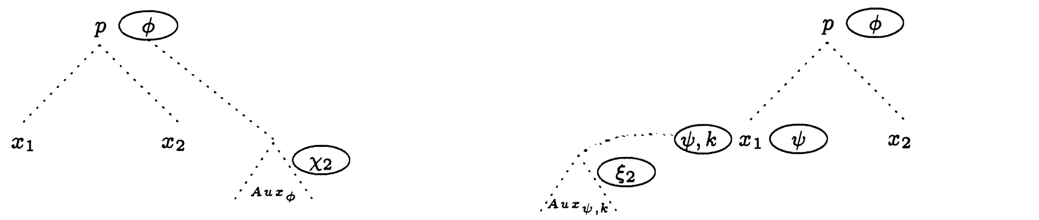
Definition. In a layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ a sub-PRS $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ is *nested* if at least one Φ_1-defining equation exists with a Φ_2-term in its right-hand side.

Theorem. In a layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ the sub-PRS $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ is nested in a PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ if and only if a Φ_1-term exits in the reduction of which a Φ_2-term occurs.

Proof. ($\Rightarrow$) by definition of nested PRS's.

($\Leftarrow$) By induction over the number of reduction steps between the Φ_1-term, and the Φ_2-term. Assume this number is 1, clearly the Φ_2-term occurs in the right-hand side of the matching equation of the Φ_1-term.

Assume this number is N and we have proved the theorem for N-1. By definition of a PRS the only equations in $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ in which Φ_2-terms occur, are Φ_2-defining equations. Moreover, in the complete specification Φ_2-terms may also occur in the right-hand side of Φ_1-defining equations. So, the term in the reduction sequence prior to the



a. $\phi(p(x_1, x_2), \ldots) = f(\xi_2(C, \ldots), \ldots)$ **b.** $\phi(p(x_1, x_2), \ldots) = \psi(x_1, f(\xi_2(C, \ldots)), \ldots))$

Figure 5: Auxiliary attributed terms

appearance of the Φ_2-term is either a Φ_1-term, in which case the theorem has been proved, or another Φ_2-term, and in that case there are N-1 reduction steps between this Φ_2-term and the original Φ_1-term. □

In the sequel we suppose a layered PRS to be well-presented and its sub-PRS's to be nested. We will write $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ for a PRS that is not contained in any other PRS, and $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ for its sub-PRS.

3.2 Auxiliary attributed terms

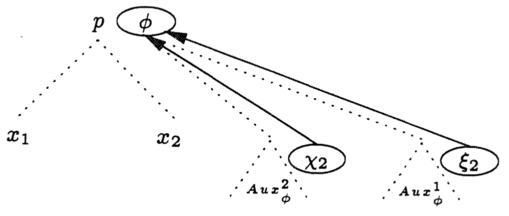
As we will use layered PRS's for incremental evaluation of Φ_1-attributes we want to store Φ_2-terms that occur in the reduction of a Φ_1-term. To this end we introduce *auxiliary attributed terms*: G_2-terms decorated with attributes that are associated with Φ_2-functions.

Let $\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$ be a Φ_1-defining equation with Φ_2-terms in τ. The first subterm of each Φ_2-term in τ, is an auxiliary term for an attribute of p. If the Φ_2-term is *not* a subterm of a Φ_1-term its first subterm is an auxiliary term for the synthesized attribute of function ϕ at the top node of p (See Figure 5a). If the Φ_2-term is a subterm of a Φ_1-term $\psi(x_i, w_1, \ldots, w_m)$ it can only be the subterm of a parameter term, say w_k. Then, the first subterm is an auxiliary term for the inherited attribute of the k-th parameter of ψ at the i-th child of the constructor. (See Figure 5b).

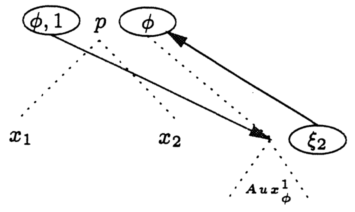
The right-hand side τ of the Φ_1-defining equation may be such that an attribute has more than one auxiliary term. This is the case in equation 2. We want each auxiliary attributed term to be related to exactly one auxiliary term in τ, so that each occurrence of the same term will be stored separately with its own attributes. We therefore distinguish the auxiliary terms of one attribute by numbering them in order of left to right occurrence in τ. Well-presentedness guarantees that the numbering of auxiliary terms for inherited attributes is consistent over the set of Φ_1-defining equations.

In a non-conditional PRS not more than one defining equation exists for each pair (ϕ, p). This equation determines the set of auxiliary terms for the associated synthesized attribute ϕ of the sort of p. In a conditional PRS, however, several defining equations may exist for one pair (ϕ, p), and the auxiliary terms for ϕ at p may be different in each equation. Since the set of auxiliary terms of ϕ at p is the union of the sets of auxiliary terms of ϕ derived from each defining equation over (ϕ, p), we will give different numbers to auxiliary terms in different equations.

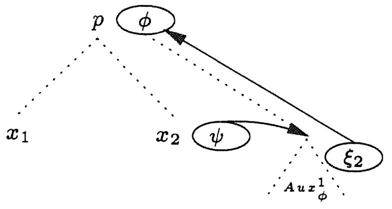
In the sequel we assume all auxiliary terms to be numbered. We write Aux^i_{att} for the i-th auxiliary term of attribute *att*, and $\{Aux_{att}\}$ to refer to the set of auxiliary terms of *att*.



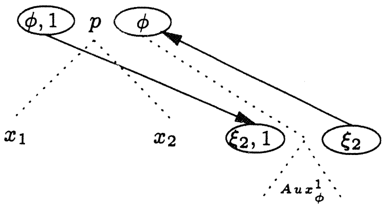
a. $\phi(p(x_1,x_2),\ldots) = f(\xi_2(C^1,\ldots),\chi_2(D^2,\ldots))$



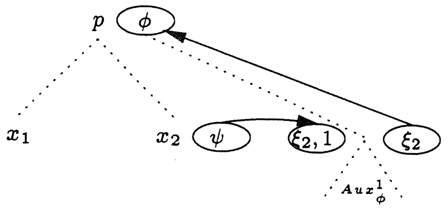
b. $\phi(p(x_1,x_2),y_1) = f(\xi_2(y_1,\ldots),\ldots))$



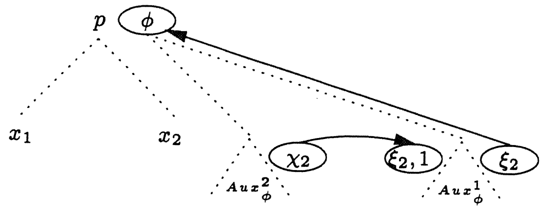
c. $\phi(p(x_1,x_2),\ldots) = f(\xi_2(\psi(x_2,\ldots),\ldots),\ldots))$



d. $\phi(p(x_1,x_2),y_1) = f(\xi_2(C^1,y_1),\ldots)$



e. $\phi(p(x_1,x_2),\ldots) = f(\xi_2(C^1,\psi(x_2,\ldots)),\ldots))$



f.$\phi(p(x_1,x_2),\ldots) = f(\xi_2(C^1,\chi_2(D^2,\ldots)),\ldots))$

Figure 6: Dependencies for auxiliary attributed terms

3.3 Attribute dependencies in a layered PRS

Attribute dependencies are needed to guide incremental evaluation. In a layered PRS attribute terms are not only Φ_1-attribute terms but also Φ_2-terms and their parameter subterms. From a Φ_1-defining equation $\phi(p(x_1,\ldots,x_n),y_1,\ldots,y_m)=\tau$ we derive dependencies $D_{\phi p}$ by analyzing τ.

1. To determine the predecessor attributes for the synthesized attribute associated with ϕ at the top node of $p(x_1,\ldots,x_n)$, take all Φ_1-terms, Φ_2-terms and y_j's in τ that are not subterms of a Φ_1- or Φ_2-term. The attributes associated with these terms are predecessors of ϕ. Figure 6 illustrates how a Φ_1 synthesized attribute can depend on a Φ_2-attribute.

2. To determine the predecessor attributes for an inherited attribute at a sub-node of $p(x_1,\ldots,x_n)$ and associated with a parameter term in τ, take all Φ_1-terms, Φ_2-terms and y_j's in the parameter term that are not subterms of a Φ_1- or Φ_2-term. The attributes associated with these terms are predecessors of the inherited attribute.

3. To determine the predecessors of an auxiliary term itself, the auxiliary term is analyzed in the same way as parameter terms in 2, (Figure 6b,c).

4. To determine the predecessors of an inherited attribute at the top of an auxiliary term, the corresponding parameter term is analyzed in the same way as parameter terms in 2, (Figure 6d,e,f).

Attribute dependencies from Φ_2-defining equations are derived in a similar fashion.

The attribute dependency graph of a G_1-term is composed by patching the dependencies of all the constructors in the term, together with the auxiliary terms, and adding the attribute dependencies in the stored auxiliary terms.

Such a dependency graph never contains cycles. As there are no (direct) dependencies between (attribute at) auxiliary terms of different attributes we can consider $\{Aux_{att}\}\cup\{att\}$ as one attribute in the dependency graph. The dependency graph we obtain this way is the one of a simple (= non-layered) PRS. This graph never contains cycles since a PRS is equivalent to a strongly non-circular attribute grammar, and the dependency graphs are exactly the dependency graphs that belong to the equivalent attribute grammar. There are no cycles within $\{Aux_{att}\}\cup\{att\}$ either. As a result of the numbering of auxiliary terms attributes of Aux^i_{att} never depend on attributes of Aux^j_{att} if $j<i$. Moreover, each auxiliary term is an attributed term of a PRS, hence does not contain cycles.

3.4 Fine-grain incremental evaluation

Suppose we have to reduce a term $\phi(T,t_1,\ldots,t_n)$ using a layered well-presented PRS $\langle G_1,S_1,\Phi_1,Eq_1,Eq_{\Phi_1}\rangle$ with sub-PRS $\langle G_2,S_2,\Phi_2,Eq_2,Eq_{\Phi_2}\rangle$. The term T is stored and decorated with attributes and auxiliary terms. Initially, these auxiliary terms do not have a value, and only the number and kind of attributes at their top nodes is known. Upon reduction normal forms of Φ_1-terms and parameter terms are stored in the appropriate attributes. An auxiliary term obtains a value a soon as the corresponding Φ_2-term occurs in the reduction. The first argument of this term becomes the value of the auxiliary term, the (normal forms of the) other arguments are stored in the inherited attributes of its top node.

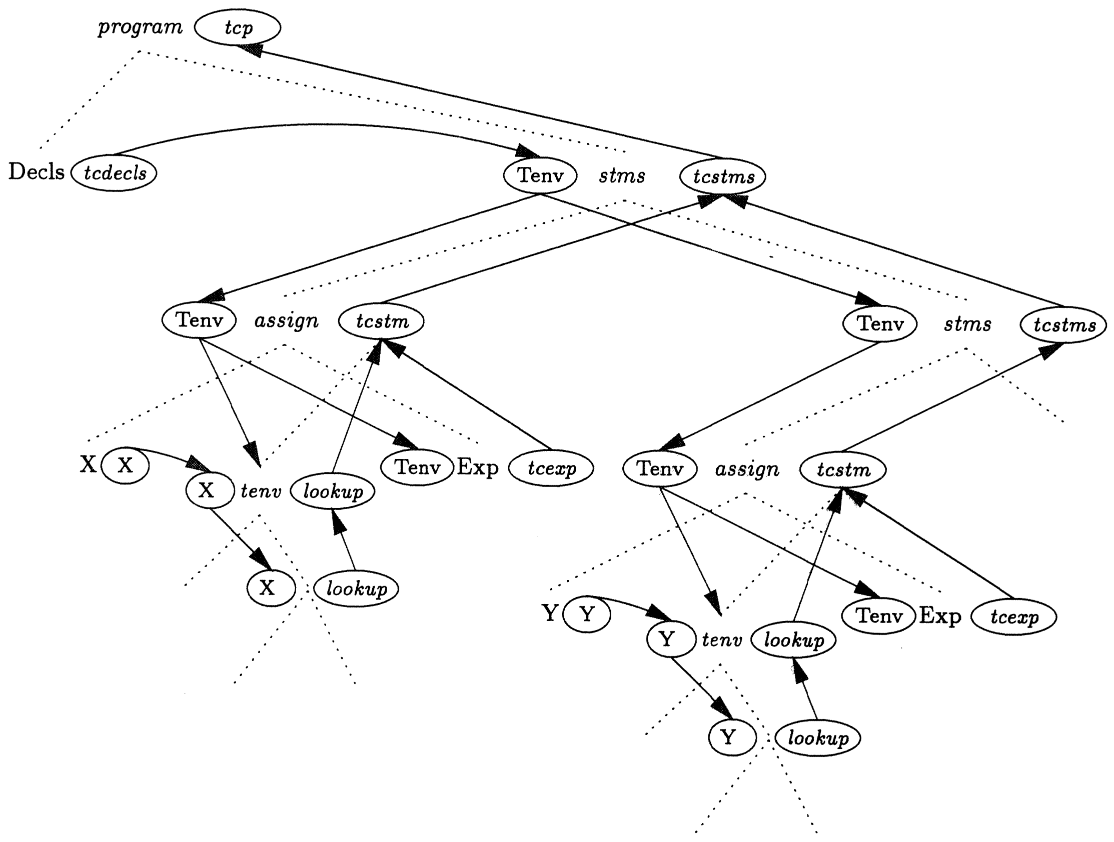


Figure 7: Part of an attributed term with auxiliary attributed terms

Once the auxiliary term has a value, its subterms can be decorated with attributes. Upon further reduction of the Φ_2-term these attributes obtain their values.

Editing the term T corresponds to a subterm replacement in the attributed term T. While updating attributes inherited attributes at the top of an auxiliary term may get a new value. The attribute updating is then applied to this auxiliary term. If the value of its top synthesized attribute changes as well, attribute updating proceeds for the successors of this attribute. Also it is possible that during attribute updating an auxiliary term itself obtains a new value. In that case we first have to determine the difference Δ between the old value and the new value. We define Δ as a pair (*newsub,path-to-root*), with *newsub* the smallest subterm of the new value, such that the new auxiliary term is obtained from the old one by replacing the subterm at the end of *path-from-root* by *newsub*[1]. Attribute updating in the auxiliary term then starts at the replacement node.

Figure 7 shows part of an attributed term with auxiliary attributed terms that belongs to the type check specification with the type environment of Figure 1 and 3.

The main drawback of this method is the calculation of Δ that has to be repeated for each auxiliary term. The specification of the type checker shows an opportunity for improving the method. From equations [Tc1],...,[Tc5] can be concluded that all auxiliary terms are copies of one and the same attribute *tcdecls*. We can use this information to improve on both storage and time, as will be explained in the next section.

4 Improving storage and speed

Copy dependencies can be derived from defining equations and can be used to replace auxiliary terms that are all copies of one and the same attribute by one multiply attributed term. Such a multiply attributed term is equipped with a hash table. We gain space as well as time by using these two. Figure 8 shows a multiply attributed type environment connected to a *program* term with type check attributes.

4.1 Copy dependencies and origin attributes

When attribute dependencies are derived from defining equations it is easy to spot the copy dependencies. For instance, in the equation

$$[\text{Tc1}] \quad \textit{tcp}(\textit{program}(\text{Decls}, \text{Stms})) = \textit{tcstms}(\text{Stms}, \textit{tcdecls}(\text{Decls}, \textit{empty-tenv}))$$

the attribute for *tcp* is a copy of the one for *tcstms* and the attribute for the parameter of *tcstms* is a copy of the *tcdecls* attribute. In equation

$$[\text{Tc4}] \quad \textit{tcstms}(\textit{stms}(\text{Stm}, \text{Stms}), \text{Tenv}) = \textit{and}(\textit{tcstm}(\text{Stm}, \text{Tenv}), (\textit{tcstms}(\text{Stms}, \text{Tenv})))$$

the inherited attributes of the sub-sorts of *stms* are copies of the inherited attribute of its top-sort.

In general attribute update procedures may benefit considerably by taking copy dependencies into account [Hoo86]. Here, we only focus on their use for auxiliary attributed terms.

[1]If we had had an attribute update algorithm for multiple subterm replacement we would, of course, have defined Δ differently.

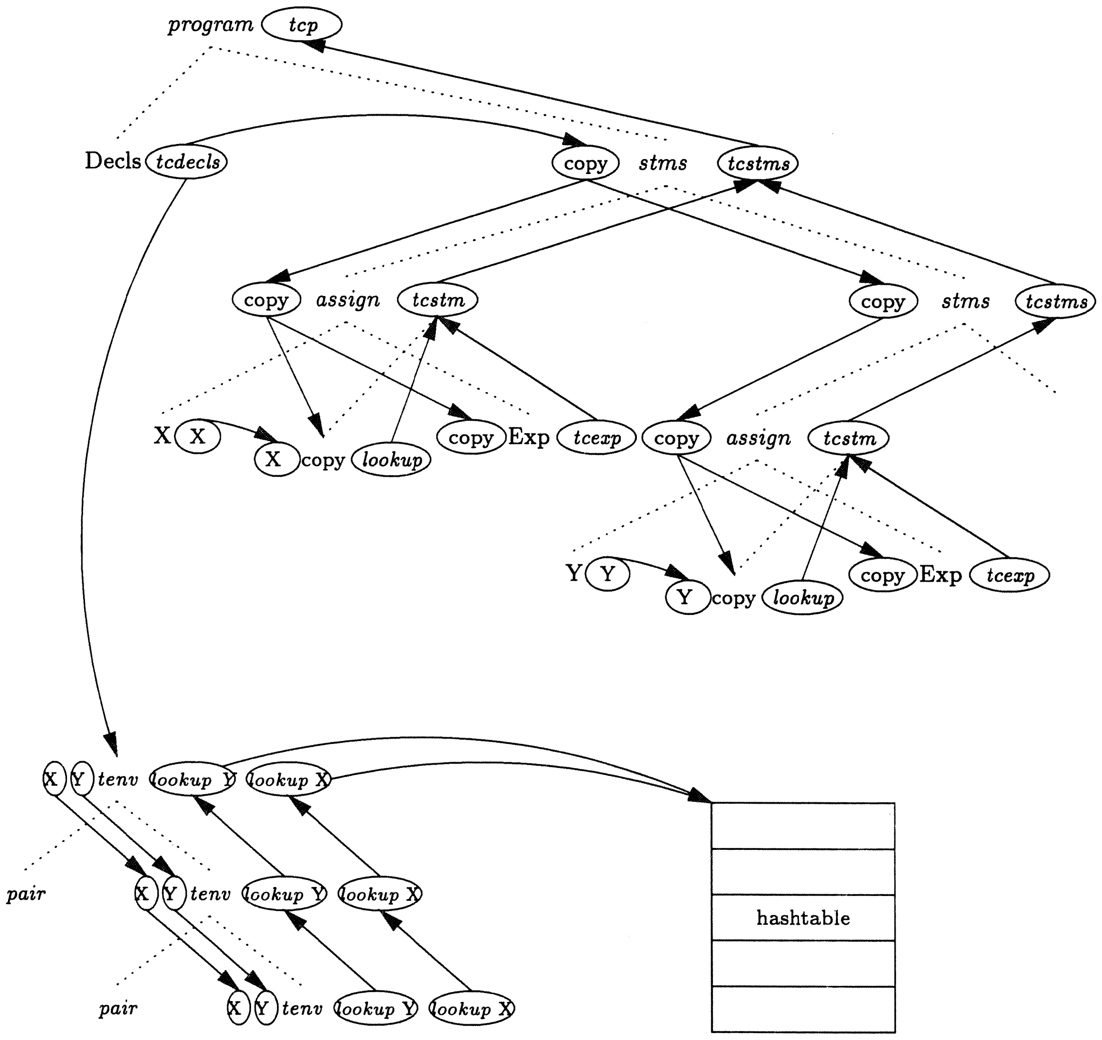


Figure 8: Multiply attributed term

In an attributed terms with auxiliary terms that are copies of their predecessor attribute we walk from the auxiliary terms backward along the copy dependencies as far as possible to find the *origin* attribute. This attribute then will be connected to a copy that will be multiply attributed with several independent dependency graphs.

4.2 Multiply attributed term with hash table

A *multiply attributed term* is a term which may have any number of attribute graphs for one incremental function. The values of the inherited attributes at the beginning of each graph, (that is, at the top node of the term) differ. All the other attributes in such a graph are therefore labeled with the values of their top inherited attributes. For example, a multiply attributed type environment has a different *Id-lookup* graph for each identifier looked for. All *Id* and *lookup* attributes are labeled with the value of the top *Id* attribute.

A hash table for attribute values at the top node is attached to a multiply attributed term. The hash-keys are determined by the name of a function and the values of the inherited attributes of this function. The entries are the values of the corresponding synthesized attribute.

4.3 Revised update procedure

During incremental evaluation a multiply attributed term obtains its value as soon as the value of the connected origin attribute is known. The decoration of the term takes place when a term $\phi_2(Aux, par_1, \ldots, par_m)$ is to be reduced. If the corresponding auxiliary term is a pointer to the multiply attributed term, Aux need not be stored. First, the normal forms of the parameters are stored in the inherited attributes of the auxiliary term. Next, the hash table is checked. If an entry for the parameters exists, then this value is the normal form of the ϕ_2-term and is stored in the synthesized attribute of the auxiliary term. Otherwise, the attributed term is decorated with a new attribute graph for these parameters, the ϕ_2-term is reduced and meanwhile the attributes in the new graph obtain a value. The normal form of the ϕ_2-term naturally will be stored in the top synthesized attribute of the new graph, the hash table is updated and the normal form is stored in the synthesized attribute of the auxiliary term.

When an origin attribute changes value, the difference Δ between the old and the new value is calculated once, and a subterm is replaced in the multiply attributed term causing an update process through each of the attribute graphs. The hash table is modified for all changed synthesized attributes at the top.

In the attributed program term, all auxiliary terms that are copies of the changed origin attribute will eventually be scheduled for re-evaluation. Assuming the parameters of the auxiliary term have not changed, re-evaluation consists of a search in the hash table.

4.4 Some order calculations

We give an impression of the results for fine-grain incrementality as compared to normal incremental evaluation, for the specification of the type checker. Suppose a term *tcp*(Program) is being reduced, and assume Program has N different identifiers in the declarations and M identifier occurrences in the statements. The *tcdecls* attribute contains a Tenv with N Id-Type pairs.

When we apply incremental evaluation without auxiliary terms, a change in the declaration causing a change in one Id-Type pair in Tenv, requires M reductions of *lookup*(newTenv,Id) terms. The length of the reduction depends linearly on the number of pairs in newTenv, it is the same for identifiers that do and those that do not occur in the old pair or the new pair. So, all *lookup* reductions together are an O(MxN) operation.

Using fine-grain incremental evaluation with a multiply attributed term a similar change in the declarations causes a O(N) calculation to determine Δ(oldTenv,newTenv). In this particular example, *lookup* attributes in at most 2 graphs have changed, namely those in the the graph of the identifier equal to the one in the old pair and in the graph of the identifier that equals the one in the new pair. Updating attributes in the multiply attributed term, then is an O(N) procedure. Modifying the hash table accordingly is assumed to take constant time. For all identifier occurrences in the statement section the hash table is visited to find a possibly new *lookup* value. Again under the assumption that such a check takes constant time, evaluating all *lookup* terms is a O(M) operation. Altogether this adds up to O(M+N).

Changing one identifier in the statements will cause a *lookup*(Tenv,newId) reduction of order O(N) if we do not use auxiliary terms. If we use fine-grain incrementality one, constant time, visit of the hash table suffices when the new identifier has been checked before, otherwise the term *lookup*(Tenv,newId) has to be reduced while the multiply attributed term is decorated with an extra attribute graph and the hash table is updated. This too is a O(N) operation but slightly more expensive than plain reduction.

5 Layered PRS and Higher-Order Attribute Grammars

Whereas in algebraic specifications the syntax and semantics of a language are described in the same formalism as the syntax and semantics of auxiliary data types, in attribute grammars the description of the attributes is in another formalism than the description of the syntax and the attribute rules. Higher-order attribute grammars [VSK89, TC90] have been developed to remove this distinction between the syntactic level and the semantic level in in attribute grammars. To this end so-called *Nonterminal attributes*, NTA's, are introduced. Nonterminal attributes can be added as extra arguments to constructors of the underlying grammar. A nonterminal attribute is either a nonterminal of this grammar which is used as a value in an attribute equation, or an attribute value, e.g, a type environment, described by means of production rules. During attribute evaluation a tree is expanded when its NTA's obtain a value.

In [CFZ82] the equivalence between strongly non-circular attribute grammars's and well-presented primitive recursive schemes with parameters has been proven. This constructive proof is the basis for the incremental implementation of PRS's. Besides, incremental implementations for higher-order attribute grammars are illustrated with an example of a type checker similar to the one we use [VSK90]. We therefore investigate the relationship between layered PRS's and higher-order attribute grammars.

We translate a *layered* PRS into a HAG. Auxiliary terms of this PRS will be translated into nonterminal attributes. The resulting HAG will be strongly non-circular. Moreover, we give a translation scheme for translating strongly non-circular HAG's into an algebraic specification. This algebraic specification is not necessarily a (layered) PRS.

Definition. A higher-order AG is strongly non-circular if for each node in each (extended) abstract syntax tree an attribute evaluation order exists that does not depend on the particular subtree rooted at that node.

5.1 Layered PRS $\Rightarrow$ strongly non-circular HAG

We translate the layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ with a sub-PRS, $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$, into a higher-order attribute grammar. The sub-PRS can be layered itself.

1. Translate $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ into a strongly non-circular attribute grammar according to the scheme of Courcelle and Franchi-Zannettacci: Synthesized attributes are associated with incremental functions and inherited attributes with the parameters of these functions. Attribute rules for the constructors of G_1 are derived from the defining equations.
2. For each occurrence of a function $\phi_2 \in \Phi_2$ in an attribute rule r of some G_1 constructor p, the sort of the first argument of this ϕ_2-term becomes a (fresh) nonterminal attribute and is added as an extra argument to constructor p. The attribute rules of p are modified in the following way.
   (a) Add a rule in which this nonterminal obtains a value, namely the subterm of the ϕ_2-term in r.
   (b) Add a rule in which the value of the inherited attributes of the nonterminal attribute is determined, namely by the term at the corresponding parameter position in r.
   (c) Replace in r the ϕ_2-term by the synthesized attribute ϕ_2 of the new nonterminal attribute.
3. Translate $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$ into a HAG and merge this HAG with the constructors and attribute rules already obtained.

If we apply this scheme to the type checker specification of Figure 1 and 3 the equation

$$[\text{Tc5}]\quad \mathit{tcstm}(\mathit{assign}(\text{Id}, \text{Exp}), \text{Tenv}) = \mathit{compatible}(\mathit{lookup}(\text{Tenv}, \text{Id}), \mathit{tcexp}(\text{Exp}, \text{Tenv}))$$

would cause the *assign* constructor to be extended with a nonterminal attribute TENV. The ID of the *assign* constructor becomes the inherited attribute of the Type environment and *lookup* its synthesized attribute. The resulting attribute rules are:

$$\mathit{assign} : \text{STM} \rightarrow \text{ID} \times \text{EXP} \times \text{TENV} \left\{ \begin{array}{l} \text{EXP}.\mathit{tenv} = \text{STM}.\mathit{tenv} \\ \text{TENV} = \text{STM}.\mathit{tenv} \\ \text{TENV}.\mathit{id} = \text{ID} \\ \text{STM}.\mathit{tcstm} = \mathit{compatible}(\text{TENV}.\mathit{lookup}, \text{EXP}.\mathit{tcexp}) \end{array} \right.$$

The translation of a layered PRS results in a HAG consisting of strongly non-circular attribute grammars. The evaluation of attributes in a node of an unexpanded G_1 term or in a nonterminal term, does not depend on the subterm rooted at that node. Moreover, as has been argued in Section 3.3, dependencies between (attributes of) nonterminal attributes never introduce cycles. Hence, the resulting HAG is strongly non-circular itself.

The resulting HAG can be called *layered* since attribute rules for Φ_2 in G_2 never contain Φ_1-attributes. Since we do not require $G_1 \cap G_2 = \emptyset$, the fact that it is layered has no implications for the *sorts* of the nonterminal attributes of G_2.

5.2 Strongly non-circular HAG $\Rightarrow$ algebraic specification

A strongly non-circular HAG can be translated into an algebraic specification using the following scheme.

1. Each synthesized attribute of a sort *Sort* is associated with a function in the algebraic specification. The first argument of that function is *Sort*. Parameters of this function are the inherited attributes of *Sort* the synthesized attribute may depend on. For each constructor

$$p : X_0 \rightarrow X_1 \ldots X_n \; NTA_1 \ldots NTA_k$$

   the constructor without its NTA's is added to the signature of the algebraic specification.

2. Defining equations are derived from attribute rules of constructor p as follows. For each rule $\phi.X_0 = \tau$ of p create an equation

$$\phi(p(x_1, \ldots, x_n, y_1, \ldots, y_m) = \tau$$

   Until there is no attribute or NTA left in τ do:

   (a) Replace every inherited attribute by the right-hand side of its defining rule.

   (b) Replace every synthesized attribute $\psi.X_i$ by a term $\psi(x_i, z_1, \ldots, z_l)$ with z_j the inherited attributes that form the parameters of ψ.

   (c) Replace every synthesized attribute $\chi.\text{NTA}_i$ by a term $\chi(\text{NTA}_i, w_1, \ldots, w_l)$ with w_j the inherited attributes that form the parameters of χ.

   (d) Replace every NTA_i by the right-hand side of its defining rule.

When we apply this scheme to a HAG that is the translation of a layered PRS, it returns a layered PRS. The only difference between the original PRS and the new one may be the number and order of parameters of the Φ-functions. Useless parameters of Φ-functions in the original PRS, that is, parameters that do not occur in the right-hand side of the defining equations, will be removed in the new PRS.

The following example shows that the translation of a strongly non-circular HAG that is not layered, results in an algebraic specification that is not necessarily a well-presented PRS. The constructor of a while statement with attribute rules to describe its evaluation is taken from an example of an *ordered*, and therefore strongly non-circular HAG in [BMV91]. The constructor *while* is extended with a nonterminal attribute STM.

$$\textit{while} : \text{STM} \rightarrow \text{EXP} \times \text{STMS} \times \text{STM} \left\{ \begin{array}{l} \text{STM}_1 = \begin{array}{ll} \textit{if} & \text{EXP}.\textit{evalexp} = \textit{true} \\ \textit{then} & \textit{while}(\text{EXP}, \text{STMS}) \\ \textit{else} & \textit{skip} \end{array} \\ \text{EXP}.\textit{value-env} = \text{STM}_0.\textit{value-env} \\ \text{STMS}.\textit{value-env} = \text{STM}_0.\textit{value-env} \\ \text{STM}_1.\textit{value-env} = \text{STMS}.\textit{evalstms} \\ \text{STM}_0.\textit{evalstm} = \text{STM}_1.\textit{evalstm} \end{array} \right.$$

The attributes of the nonterminal attribute are equal to the ones of the other nonterminals. Applying the above translation scheme with an extra rule for translating conditions in rules into conditions for equations, we obtain the following equations for an algebraic specification.

$$\frac{\mathit{evalexp}(\text{Exp}, \text{Stms}), \text{Value-env}) = \mathit{true}}{\mathit{evalstm}(\mathit{while}(\text{Exp}, \text{Stms}), \text{Value-env}) = \mathit{evalstm}(\mathit{while}(\text{Exp}, \text{Stms}), \mathit{evalstms}(\text{Stms}, \text{Value-env}))}$$

$$\frac{\mathit{evalexp}(\text{Exp}, \text{Stms}), \text{Value-env}) \neq \mathit{true}}{\mathit{evalstm}(\mathit{while}(\text{Exp}, \text{Stms}), \text{Value-env}) = \mathit{evalstm}(\mathit{skip}, \mathit{eval}(\text{Stms}, \text{Value-env}))}$$

Both equations are correct but do not belong to a layered PRS. Since the *evalstm*-functions in the right-hand sides are equal to the one in the left-hand side their first arguments will not be classified as auxiliary terms. Neither do these equations belong to a non-layered PRS. Since the first arguments of incremental terms are not subterms of the *while*-term in the left-hand side, the strictly decreasing property of PRS's (Appendix A, (v)) is violated.

6 Conclusions and future plans

For algebraic specifications in the class of layered primitive recursive schemes a fine-grain incremental implementation is derived. The implementation is improved by taking copy dependencies between attributes into account. One of the applications of this method is that it provides an incremental implementation of the *lookup* operation on symbol tables in a type checker.

We aim at an implementation of algebraic specifications that allows incremental handling of symbol tables for different languages with a variety of scope rules. The proposed technique provides one aspect of this implementation. Further investigation of copy dependencies and constructor dependencies, may not only improve the efficiency of the general attribute update algorithm, but also lead to the incremental construction and updating of symbol tables.

The ASF+SDF meta-environment is a programming environment generator [Kli91]. From a language definition written in the algebraic specification formalism ASF+SDF [BHK89, HHKR89], a program environment is generated, currently consisting of a syntax-directed editor which is coupled to a term rewriting system. The fine-grain incremental evaluation technique will be implemented as part of this term rewriting system.

Acknowledgements

Paul Klint suggested to apply incremental techniques to symbol tables. Mark van den Brand and Pum Walters suggested the use of a hash table. Harald Vogt answered most of my questions about higher order attribute grammars. Comments by Jan Heering and Arie van Deursen improved the readability. Thank you all.

References

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

[BMV91] M.G.J. van den Brand, E.A van der Meulen, and H.H. Vogt. Integrating syntax and semantics in language specifications. Draft, 1991.

[CFZ82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I and II. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

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

[Fri88] P. Fritzson. Incremental symbol processing. In *Proceedings of the 2nd CCHSC workshop*, volume 371 of *Lecture Notes in Computer Science*, pages 11–38. Springer-Verlag, 1988.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hoo86] R. Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–25. ACM, 1986.

[HT86a] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50. ACM, 1986. Appeared as SIGPLAN Notices 21(7).

[HT86b] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, 1986.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[TC90] T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environment. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 197–208. ACM, 1990.

[VSK89] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 131–145, 1989. Appeared as SIGPLAN Notices 24(7).

[VSK90] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. On the efficient incremental evaluation of higher order attribute grammars. Technical Report CS-90-36, Utrecht University, Utrecht, 1990.

[YS91] D. Yellin and R. Strom. INC: a language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–237, 1991.

A Primitive Recursive Schemes

We first give the definition of a well-presented primitive recursive scheme with parameters. The definition is similar to the one given by Courcelle and Franchi-Zannettacci [CFZ82], except that we do not require the signatures of the "grammar" and the "semantic data types" to be disjoint. Next, we give the definition of a conditional primitive recursive scheme.

A.1 Well-presented primitive recursive schemes with parameters

(i) A PRS is indicated as a 5-tuple $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$, with G and S two signatures, and Φ a set of function, such that $\Phi \cap (G \cup S) = \emptyset$. The signature of a PRS is the union of G, S and Φ.

Eq is the set of equations over terms defined by S, Eq_Φ contains so-called Φ-*defining* equations. G and S need not be disjoint but equations over G-terms do not exist.

Intuitively, G represents a set of free constructors, describing, for instance, the abstract syntax of a programming language. Φ is a set of operations on these syntax describing, for instance, the type checking of programs. $S \cup Eq$ is the specification of the auxiliary data types. For elements of Φ and Eq_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.

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

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{3}$$

with τ a $\{x_1, \ldots, x_n, y_1, \ldots, y_m\} \cup S \cup \Phi$ term

(iv) All x_i and y_j in the left-hand side of equation 3 are different variables.

(v) The first argument of a Φ-term in the right hand side of a defining equation should be a direct sub-term of the first argument of the left hand side of that equation, that is, some x_i

A PRS is *well-presented* if all parameter terms for each constructor in G are defined uniquely. That is, if a PRS has properties (vi)-(viii) below.

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

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \qquad \phi(q(u_1, \ldots, u'_n), z_1, \ldots, z_m) = \tau',$$

the variable lists $\langle y_1, \ldots, y_m \rangle$ and $\langle z_1, \ldots, z_m \rangle$ are identical;

(vii) In all occurrences of the form

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

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 defining equations, in all sub-terms

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

of right-hand sides of defining equations of the same abstract tree constructor $p \in G$ v_j is identical to w_k.

A.2 Conditional PRS's

In a conditional PRS conditions can be added to Φ-defining equations and for each pair ϕ, p several defining equations can exist. In a conditional defining equation

$$\frac{\lambda_1 = \rho_1, \ldots, \lambda_k = \rho_k}{\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau}$$

Like τ, terms in the λ_i's and ρ_i's are $\{x_1, \ldots, x_n, y_1, \ldots, y_m\} \cup S \cup \Phi$ terms and they have property (v).

A conditional PRS is *well-presented* if (vi), (vii) and (viii) hold for terms in τ as well as λ_i and ρ_i, $i \in \{1, \ldots, k\}$.