Incremental Rewriting

Incremental Rewriting

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr P.W.M. de Meijer
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op vrijdag 7 januari 1994 te 13.30 uur

door

Emma Anna van der Meulen
geboren te Emmer-compascuum

Promotor: prof. dr. P. Klint.
Faculteit: Wiskunde en Informatica.

Printed by CopyPrint 2000, Enschede, The Netherlands.

Partial support for the research described in this thesis was received from the European Community under Esprit projects 348 (GIPE - Generation of Interactive Programming Environments) and 2177 (GIPE II - Generation of Interactive Programming Environments II), as well as from the Netherlands Organization for Scientific Research - NWO, project Incremental Program Generators.

Acknowledgements

First and foremost, I wish to extend my thanks to Paul Klint, who made it possible to perform the research presented in this thesis, both in his role as promotor and as leader of the "GIPE" group. His enthusiasm and abundance of ideas have been very inspiring. He came up with the topic of incremental typechecking and throughout my research he has provided me with invaluable suggestions. I especially want to thank him for his support in the last period of this work, and his careful reading of numerous drafts of this thesis.

Discussions with Jan Heering have been invaluable to the development of the research presented in this thesis. He helped in finding the proper literature and had a knack for asking the right questions. I am very grateful to him for his continuous interest and his meticulous reading of my work.

Arie van Deursen has always been on hand to discuss ideas and read my papers. He was never short of encouragement and friendly criticism.

Thanks are due to Casper Dik for his assistance with LeLisp and for his cooperative effort in realizing the implementation.

Thanks to Dinesh for useful comments on parts of the manuscript.

Carol Orange corrected my English and helped out with practical things.

I thank the members of the reading committee Jan Bergstra, Gregor Engels, Thomas Reps, and Doaitse Swierstra for their willingness to review this thesis. Their ample constructive remarks are gratefully acknowledged.

It has been a great pleasure to work in the stimulating and friendly atmosphere of the "GIPE" group at CWI and UvA. I thank my former and present colleagues in this group: Paul Hendriks, Jan Rekers, Pum Walters, Hans van Dijk, Wilco Koorn, Casper Dik, Arie van Deursen, Susan Üsküdarlı, Dinesh, Frank Tip, Jasper Kamperman, Mark van den Brand, Huub Bakker, Job Ganzevoort, Paul Vriend, and Eelco Visser.

Finally, I want to thank my friends and family for good times and encouragement. In particular I would like to mention Carol Orange, Casper Dik, Sjouke Mauw, Chris Prast, Annemarie Woudstra, Riek ter Haar, Hubert de Boer and, of course, my parents.

Contents

Chapter 1

Introduction

1.1 Incremental computation

Optimization techniques allow the efficient implementation of a large set of software applications for which computers would otherwise be too slow. Incremental methods are a significant class of optimization techniques, which are used in a wide variety of applications. An incremental computation stores the results of subcomputations, and reuses the results rather than recomputing them again. The following examples illustrate how this idea is applied in practice.

1.1.1 Examples

Incremental computation is especially effective when a complex computation is performed repeatedly on slightly different inputs, or when computations are performed over large data structures, to which relatively small modifications are made frequently.

Spreadsheets A standard example of a program based on incremental computation is a spreadsheet program. A spreadsheet is a rectangular array of cells containing values. A spreadsheet program specifies the formulas to compute the value of cells from values of other cells. Some cells have *input* values, values that do not depend on values in other cells. If such an input value is changed, in a straightforward implementation all values would be computed anew in a non-incremental computation. If computation is incremental, only the value of cells depending on the modified initial value are recomputed.

Text formatting Incremental text formatting is used in so-called WYSIWYG[1] editors. In these editors the text is displayed according to the layout information added by the user. During editing the text is constantly being reformatted. For large texts this is only feasible if formatting is performed incrementally. For instance, after a line has been added to a paragraph, there is often no need to redo the formatting of other paragraphs.

[1] What You See Is What You Get

Recursive functions Recursively defined functions may cause the same calculation to be repeated many times over. A classical example of a recursively defined function is the Fibbonnacci function.

$fib(n) = \textsf{if } x \leq 2 \textsf{ then } 1 \textsf{ else } fib(n-1) + fib(n-2)$

In a straightforward implementation, computation of the Fibonnaci value of an integer n requires two separate evaluations of $fib(n-2)$, three evaluations of $fib(n-3)$ and so on. Hence the cost of the computation is exponential in n. When the results for each $fib(n)$ are stored so that they can be retrieved easily, computation of $fib(n)$ is linear in n.

Programming environments Another example in which incremental computation is mandatory can be found in interactive programming environments. A simple programming environment may consist of a syntax-directed editor and a compiler. More sophisticated environments provide various tools for, for instance, type checking, pretty printing, program analysis, translation and evaluation.

A syntax-directed editor checks whether a text in an editor is correct w.r.t. the syntax of a language. Incremental syntax checking permits fast checks while the text is being modified, as in general only changed parts need to be checked. When a tool is applied incrementally, the results of intermediate computations are stored, so that when the program is edited and the tool is applied to the modified program these results can be used to avoid performing the same computations again.

1.1.2 Discussion

The performance improvements which can be gained by applying incremental techniques come at a price. Specifically, additional memory is required to store the results of subcomputations. Furthermore, the development and implementation of incremental algorithms is more complex than that of their non-incremental counterparts. Since the overall speed of computers has increased steadily due to consistent hardware improvements, the question may arise as to whether the use of incremental techniques is justified.

Indeed, when computer speed has increased sufficiently, some applications may no longer require optimization. However, regardless of the speed of computers, there will always be applications for which their performance is insufficient. This is because the increase in computing speed has continually inspired the development of more computationally intensive applications.

The overall increase in processor speed will, therefore, not obviate the need for incremental methods to improve the performance of many applications.

1.2 Incremental methods

Many techniques for incremental computation have been developed. Some are incremental algorithms for specific problems, others are of a general nature. An overview is given by Ramalingam and Reps in [RR93]. General techniques differ in the way they store information and maintain the storage, and in the granularity of incremental computation. Another important distinction is between methods based on selective recomputation and those based on finite differencing.

When results of computations are stored for incremental computation, the stored information must be structured such that it permits fast retrieval of information during subsequent computations. Related to this is that the amount of stored information must be kept within limits. Many techniques, but not all, only remember the information applicable to the last computation.

We call an incremental method *coarse-grain* if a subclass of computations exists that are inherently not incremental. For example, in spreadsheets incrementality is based on identifying exactly those cells of which the value has to be recomputed. The method is coarse-grain because computation of the new value itself is not incremental. *Fine-grain* methods on the other hand, have no principle limitations for the class of computations that are performed incrementally.

In order to explain the difference between selective recomputation and finite differencing we consider $f(\mathit{input})$, an incremental operation on *input*. When $f(\mathit{input})$ is computed intermediate results are stored. Assume that subsequently the result of the same operation is requested but for a slightly different input: $\mathit{input} + \Delta \mathit{input}$. In methods based on selective recomputation, the intermediate results for computing $f(\mathit{input})$ are used to avoid common subcomputations for $f(\mathit{input} + \Delta \mathit{input})$. In approaches based on finite differencing, the existence of a *difference* function, f_Δ is presumed. f_Δ is used in computing the output for $\mathit{input} + \Delta \mathit{input}$ by updating the previous output:

$$f(\mathit{input} + \Delta \mathit{input}) = f(\mathit{input}) \oplus f_\Delta(\Delta \mathit{input})$$

We first discuss some general methods for incremental computation, then we focus on incremental computation in generated programming environments.

1.2.1 General incremental methods

Finite differencing The term finite differencing has been coined by Paige and Koenig in [PK82]. Their ideas have been applied by Paige in his paper on programming with invariants [Pai86]. In programming with invariants, programs are transformed into incremental programs by replacing functions by an incremental counterpart taken from a library. For each incremental function f, a so-called invariant, i.e. a value $E = f(x_1, \ldots, x_n)$, is maintained during execution of the program. Whenever the value of one of the arguments, x_i changes, the value for E is updated by means of a difference function $dx_i f$.

The amount of stored information is limited as only the last value for E is preserved. The method is fine-grain, because it deals with incrementality of a single function.

INC INC is a language for incremental computation, designed by Yellin and Strom [YS91], and based on finite differencing. The language INC consists of a number of predefined functions for which difference functions exist. An INC program is a network of predefined functions in which the output of one function serves as the input for the next one. When an INC program is executed with a certain input *input*, both the input and the output values of each function application in the program are preserved. When subsequently the program is executed with new input $input + \Delta input$, the first function f of the program uses f_Δ to compute both the new output and the difference with the previous output. The output difference is passed on to the next function.

Only information for the last computation is preserved. The authors refer to this method as fine-grain, since each evaluation step is performed incrementally.

Function caching Function caching or memoizing, introduced by Michie in [Mic68], is a technique for storing results of function applications, and re-using them when a function is invoked a second time with the same arguments.

The technique is often used to improve the performance of recursive functions. An overview of various memoizing methods for recursive functions is given by Bird in [Bir80].

Pugh and Teitelbaum [PT89] have described how to use function caching for incremental evaluation for data structures like sequences and sets. In order to obtain efficient incremental algorithms the data structures must be designed in a special way so that computations over similar instances of the data type share many subcomputations.

Function caching is conceptually simple but difficult to implement efficiently. One difficulty is to find ways for fast searching through the cache. When functions are invoked with large arguments, it tends to be very expensive to compare arguments of the pending function call with the entries in the cache. Another difficulty is to keep the size of the cache within limits. Purging techniques are needed that retain the results that are likely to be reused in later computations, and remove other information.

The method is fine-grain, since the results of any function can be cached.

Incremental rewriting In the technique for incremental rewriting as proposed recently by Field in [Fie93], replaceable subterms are designated before a term T is rewritten. When the term is rewritten, the context terms in T of the designated subterms are rewritten simultaneously. The result of incremental rewriting is the normal form of the term as well as the normal forms of all context terms. They are grouped together in a term, from which each individual normal form is easily retrieved.

When a second term T' is to be reduced, and T' differs from T in one or more designated subterms, the normal form of the common context term is used as a starting point for incremental reduction of T'. The resulting term will then be added to the existing storage.

This technique for incremental rewriting is fine-grain, because there are no restrictions on the kind of functions that can be rewritten incrementally.

1.2.2 Incremental methods in generated programming environments

The following incremental methods are especially used in programming environment generators. A programming environment generator is a program which derives a programming environment for a language from the specification of the syntax and semantics of that language.

As incremental implementations in generated programming environments are to be deduced for user-defined operations, methods based on selective recomputation are better suited than methods based on finite differencing, which would require the definition of additional difference operations for each user defined operation.

Attribute grammars An attribute grammar is a context-free grammar extended with attributes for each non-terminal that describe the semantic features of symbols in the grammar. Attribution rules define the value of attributes in a production as a function of the values of other attributes. It is the most well-known manner of specifying static semantics of a language.

We classify methods for incremental attribute evaluation according to their way of storing information. Some techniques make use of the relation between attribute grammars and functional languages by implementing an attribute grammar as a functional program and use sophisticated transformation techniques for incremental evaluation. See for instance work by Pennings, Swierstra, Vogt and Kuiper [PSV92, VSK90, Vog93].

In most programming environments derived from attribute grammars, however, incremental attribute evaluation is based on annotating the abstract syntax tree of a program with attributes in which semantic values can be stored. When the program is edited, subtrees in the abstract syntax tree are replaced, hence information relevant only to the old parts is automatically removed. Attributes in the new subtrees must be evaluated, as well as the attributes whose value depends on the values of the new attributes.

This method is coarse-grain because the re-computation of an individual attribute value is not incremental.

Reps et al. [Rep82, RTD83] describe incremental attribute update algorithms based on this way of storing and updating intermediate results. The strategy is implemented in the Synthesizer Generator [RT89a, RT89b].

Extended Affix Grammars Extended affix grammars [Kos91] are a language specification formalism closely related to attribute grammars. Incremental evaluation of extended affix grammars is used in the environment generator Pregmatic described by van den Brand in [Bra90, Bra92]. A special feature in his implementation is that attribute values are evaluated (incrementally) during parsing rather than afterwards. Attribute values are used to direct parsing, to solve ambiguities, and to stop parsing when an error is found.

Context relations Context relations, developed by Snelting and Henhapl [Sne91, HS84] are another formalism for specifying static semantics of languages. They are used in the PSG system, described by Bahlke and Snelting in [BS86]. Context relations have been developed as an alternative to attribute grammars with the aim to be able to check arbitrarily incomplete programs.

A specification of a type checker describes for each abstract syntax constructor the relation between the type of the constructor and those of its subtrees. When a program in an editor is checked, a data-base, *context relation*, is maintained which contains for each node the list of all the types it may have in the given context. A program is incorrect if the entry for a certain node is empty. When the program is edited the data-base is updated incrementally by unifying it with the context relations for the new subtree.

Typol Typol is a specification formalism used in the Centaur system [Des88, DT89, BCD+89]. It is based on natural semantics [CDD+85, Kah87] and is closely related to PROLOG.

Evaluation in the Typol setting means building a proof tree. In [ACG92], Attali, Chazarain and Gillette indicate a class of Typol specifications for which incremental functional evaluators can be generated, which decorate the proof tree with attributes. In this particular class of Typol specifications, a natural relation exists between parts of the abstract syntax tree of the program in the editor and parts of the proof tree. When the program is edited, the corresponding parts of the proof tree are replaced, and subsequently the attributes of the new parts of the proof tree and of all attributes that depend on them are evaluated.

1.3 Incremental type checking

Because incremental type checking will be used as an example throughout the thesis, we consider it in a little more depth.

A standard way of type checking a program consists of two parts. First, collect all declaration information by traversing the declarations one by one. Then, visit each statement and verify that the variables that occur in it have been declared and are being used in a type correct manner.

Incremental type checking is intended for the situation that a program is to be type checked after a modification has been made to it. How can the results of type checking the original program be used for the incremental type checking of the modified program? Figure 1.1 shows two examples of program modifications.

When $\boxed{Z := Z + \mathsf{1}}$ is a modified statement, it is clear that only this statement has to be rechecked. For this check we can use the previously saved declaration information.

However, when $\boxed{Y : \mathsf{natural}}$ is a modified declaration, potentially all statements in the program will have to be checked. One of the challenges of incremental type checking is to find general methods that limit the recomputation to essential parts, e.g, in this example, the statements in which Y actually occurs.

```
begin
   X : natural,
   Y : natural,
   Z : natural;
  read X;
  Y := 1;
  Z := 1;
  if (X ≠ 0)
    then
      while (Z ≠ X) do
          Z := Z + 1;
          Y := Y * Z
      end
  fi;
  write Y
end
```

```
begin
   X : natural,
   Y : natural,
   Z : natural;
  read X;
  Y := 1;
  Z := 1;
  if (X ≠ 0)
    then
      while (Z ≠ X) do
          Z := Z + 1;
          Y := Y * Z
      end
  fi;
  write Y
end
```

Figure 1.1: A program for computing the factorial of a number

1.4 This thesis

1.4.1 Context

The ASF+SDF Meta-environment [Kli91] is a programming environment generator based on the algebraic formalism ASF+SDF [BHK89b]. From a language specification in ASF+SDF an environment is generated consisting of a syntax-directed editor for the language in question, as well as a collection of tools that have been specified. Typically, these tools include type checkers, pretty printers, compilers and evaluators.

An algebraic specification consists of a signature and a set of equations. The signature introduces sorts and functions over these sorts and thus defines a set of *terms*. The equations specify equivalences between the terms of the signature. Many algebraic specifications can be made executable by interpreting the equations as a term rewriting system. The equations are then oriented from left to right.

From the definition of the syntax of a language a syntax directed editor is generated. Such an editor checks whether a text is a syntactically correct program. If this is the case an *abstract syntax term* Program of the text is created and updated during later editing.

Tools for a language are specified by means of functions over programs and their substructures. Equations define the behaviour of these functions. If a tool specified by means of the function tool is to be applied to a program in an editor a term tool(Program) is created and reduced according to the generated rewrite system.

The prime motivation for the research described in this thesis is the desire to have incremental tools in the generated environments. When a program is edited and a tool is applied again to the modified program Program′ reduction of tool(Program′) must make use of intermediate results of the previous reduction of tool(Program). We therefore need a technique for *incremental rewriting*.

1.4.2 Method

A naive approach to incremental rewriting is to store all terms that occur in a reduction together with their normal forms. Clearly, as with naive function caching, this will result in a large, ill-structured cache, and requires additional methods for efficient comparison of stored information and terms to be reduced, as well as techniques for selecting the information that can be removed.

We have found a relatively simple solution for these problems by restricting ourselves to a subclass of algebraic specifications, namely the class of primitive recursive schemes. These are equivalent to attribute grammars. Primitive recursive schemes are well-suited for specifying tools like type checkers, pretty printers and compilers. We use the equivalence between algebraic specifications and attribute grammars to select functions which can be implemented incrementally. These functions typically apply to a program or a substructure of a program, and we can therefore store the results of applying them in attributes of the parse tree of the program.

Due to the structure of the specifications, the cache (parse tree) can be traversed during incremental rewriting, so that relevant information is retrieved efficiently. Obsolete information is automatically discarded when the program is edited.

Like other attribute-grammar-based methods, our basic method is coarse-grain. This is known to be inefficient when an attribute which contains a large aggregate value, like a set or a table, is altered. We exploit the uniform nature of algebraic specifications by extending the technique for incremental rewriting to functions over auxiliary data types used by incremental functions. In particular functions over these aggregate values can be made incremental. This extension yields a fine-grain incremental rewriting technique, that solves for instance the problem of incremental type checking sketched in Section 1.3.

1.4.3 Achievements

The main contribution of this thesis are the design and implementation of algorithms for coarse-grain and fine-grain incremental rewriting.

1.4.4 Overview

In Chapter 2 we develop a technique for incremental rewriting for primitive recursive schemes, based on storing normal forms in attributes of a parse tree. The technique is capable of dealing with multiple subtree replacements.

A specification in ASF+SDF may contain conditional equations. Although the presence of conditions does not add to the formal power of the formalism, it certainly adds flexibility and user-friendliness. One could apply the technique for incremental rewriting to a specification with conditional equations by transforming it into a specification with non-conditional equations, and then apply the technique described in Chapter 2. This transformation would lead to a less efficient rewrite system. Instead, we discuss in Chapter 3 efficient algorithms for conditional incremental rewriting, based on dynamic updating of the attribute dependency graph.

Lists frequently occur in the description of the syntax of programming languages. In Chapter 4 we study incremental implementation of functions on lists. Earlier research has addressed the problem with finite differencing methods.

We describe a class of list functions, as well as a method for efficient incremental evaluation by selective recomputation.

In Chapter 5, we introduce the class of layered primitive recursive schemes and describe a fine-grain incremental implementation for specifications in this class.

The techniques developed in Chapters 2–5 have been implemented as an extension of the rewrite system of the ASF+SDF meta-environment. In Chapter 6 we describe the implementation of incremental rewriting, as well as the interfaces with the standard term rewriting machine and the editors. The performance of this implementation has been evaluated and the results are reported in Chapter 7.

In Chapter 8 we conclude with a summary, an account of various experiences with the implementation, a discussion of limitations and suggestions for further research.

Chapter 2

Incremental rewriting for algebraic specifications

We present a technique for deriving incremental implementations for a subclass of algebraic specifications, namely, well-presented primitive recursive schemes with parameters. We introduce a concept adapted from the translation of well-presented primitive recursive schemes to strongly non-circular attribute grammars. We store 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 class of well-presented primitive recursive schemes is a very natural one for specifying the static semantics of languages.

2.1 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 [CFZ82]. The result of each function is interpreted as a synthesized attribute of the first argument of this function and the parameters of this function are interpreted as the inherited attributes of this sort. 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 use attributed trees for storing normal forms for incremental evaluation of terms.

Our strategy is to be implemented as part of the term rewriting engine of the ASF+SDF Meta-environment [Kli93b], a programming environment generator. From a language definition written in the algebraic specification formalism ASF+SDF, a programming environment is generated. The main component of a generated environment is a syntax-directed editor connected to a term rewriting system for evaluating terms in the editor.

Incremental evaluation in this context concerns the application of functions ϕ to the abstract syntax tree T of a text in an editor. During reduction of the term $\phi(T)$ normal forms are stored in attributes of T. When the text is edited subtrees are replaced in the abstract syntax tree T. After each subtree replacement attributes whose value depend on the replaced subtree are marked as "unreliable". Let T' be the result of subtree replacements in T. If subsequently the term $\phi(T')$ is reduced the values of reliable attributes can be used to avoid reduction steps, whereas other attributes obtain a new type check value.

Because we do not wish to intrude on the user while editing we aim at keeping the process of marking attributes as simple and as efficient as possible. For the same reason re-evaluation of the ϕ value of the modified text takes place only on explicit request by the user. This is possible because our algorithms for incremental evaluation support *multiple subtree replacements*.

Our strategy for incremental evaluation can be thought of as leftmost innermost rewriting with short cuts and side effects for updating attributes. This makes it easy to combine incremental evaluation and non-incremental evaluation.

Overview of this chapter

In Section 2.2 we give definitions and examples for algebraic specifications and attribute grammars, and we explain the basic steps of incremental attribute evaluation strategies. Section 2.3 defines a subclass of algebraic specifications called well-presented primitive recursive schemes and explains the construction of an attribute grammar from a well-presented primitive recursive scheme. Sections 2.4 to 2.7 constitute the heart of the chapter. Section 2.4 presents a description of the storage for incremental evaluation. In Section 2.5 we describe our method for incremental evaluation of terms with respect to a primitive recursive scheme. In Section 2.6 we prove the correctness of this method. Section 2.7 discusses various properties of the approach. Sections 2.8 and Section 2.9 present related work and conclusions.

2.2 Preliminaries

2.2.1 Algebraic specifications

Definition 2.1 An algebraic specification $\langle \Sigma, E \rangle$ consists of a signature Σ in which sorts are defined as well as functions over these sorts, and a set of equations E over terms defined by the signature. $\square$

The equations in an algebraic specification define equivalence classes of the terms that can be constructed from the signature. Algebraic specifications can be given for any abstract data type, and can be used to describe the abstract syntax as well as the static and dynamic semantics of languages.

Example 2.1 Figure 2.1 presents part of the type checker of a simple programming language in the algebraic specification formalism ASF [BHK89a]. It will serve as our running example. Each program consists of a (possibly empty) series of variable sorts: PROGRAM DECLS DECL STMS STM BOOL TENV

functions:

program	: DECLS $\times$ STMS		$\rightarrow$ PROGRAM
empty-decls	:		$\rightarrow$ DECLS
decls	: DECL	$\times$ DECLS	$\rightarrow$ DECLS
stms	: STM	$\times$ STMS	$\rightarrow$ STMS

tcp	: PROGRAM		$\rightarrow$ BOOL
tcdecls	: DECLS	$\times$ TENV	$\rightarrow$ TENV
tcdecl	: DECL	$\times$ TENV	$\rightarrow$ TENV
tcstms	: STMS	$\times$ TENV	$\rightarrow$ BOOL
tcstm	: STM	$\times$ TENV	$\rightarrow$ BOOL

variables:

$Decls$	$:\rightarrow$ DECLS	$Decl$	$:\rightarrow$ DECL
$Stms$	$:\rightarrow$ STMS	Stm	$:\rightarrow$ STM
Env	$:\rightarrow$ TENV		

equations:

[Tc1] tcp(program($Decls$,$Stms$)) = tcstms($Stms$,tcdecls($Decls$,empty-env))

[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($Stms$, $Tenv$))

Figure 2.1: Part of the algebraic specification of a type checker

declarations followed by a series of statements. The functions tcp, tcdecls, tcdecl, tcstms and tcstm are introduced to specify the type checking of, respectively, programs, declarations and statements. The equations describe how a program is type checked. According to equation [Tc1] the result of type checking a program equals the type checking of its statements given the result of type checking the declarations. Type checking the declarations yields a type-environment, a table of variables and types, indicated by the sort TENV. Equation [Tc2] formalizes that type checking the empty declaration in the presence of a type-environment returns that same type-environment. [Tc3] describes the type checking of a list of declarations. The result of type checking the first declaration is used to type check the remaining declarations. The result of type checking a list of statements is a Boolean value: the conjunction of type checking its first statement and type checking the rest of the remaining statements ([Tc4]). □

Incremental evaluation

Many algebraic specifications can be 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 the term.

The idea of incremental evaluation of terms is that while evaluating a term we make use of the normal forms of terms that have been stored during a previous reduction of a slightly different term. The naive 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 whether a term has been reduced before.

We are looking for a structure in which normal forms can be stored so that they can easily be retrieved and updated. We turn to attribute grammars because attribute grammars do provide such a structure and Courcelle and Franchi-Zannettacci have proved a correspondence between attribute grammars and a subclass of algebraic specifications [CFZ82].

2.2.2 Attribute grammars

Definition 2.2 An attribute grammar $\Gamma = \langle G, ATT, R, I \rangle$ consists of a signature G, defining sorts and abstract tree constructors, extended with an attribute system. For each sort X of G two disjoint sets of attributes are defined: the *inherited attributes*, INH(X), and the *synthesized attributes*, SYN(X). ATT is the union of all inherited and synthesized attributes: $ATT = \bigcup_{\mathrm{X}} \mathrm{INH(X)} \cup \mathrm{SYN(X)}$. To each abstract tree constructor $p : X_1 \times \ldots \times X_n \rightarrow X_0$ of G, attribute definition rules R_p are added for specifying the values of the synthesized attributes of the parent X_0 and the values of the inherited attributes of the children $X_i, i \leq 1 \leq n$. $R = \bigcup_{p \in G} R_p$. The interpretation I indicates the domain of attribute values. □

Example 2.2 Figure 2.2 shows the same type checker as Figure 2.1 but now written as an attribute grammar. Attributes and semantic rules are added to the four sorts: PROGRAM DECLS DECL STMS STM

attributes:

INH(PROGRAM)	$= \emptyset$	SYN(PROGRAM)	$= \{\mathsf{tcp}\}$
INH(DECLS)	$= \{\mathsf{Tenv}_{\mathsf{DECLS}}\}$	SYN(DECLS)	$= \{\mathsf{tcdecls}\}$
INH(DECL)	$= \{\mathsf{Tenv}_{\mathsf{DECL}}\}$	SYN(DECL)	$= \{\mathsf{tcdecl}\}$
INH(STMS)	$= \{\mathsf{Tenv}_{\mathsf{STMS}}\}$	SYN(STMS)	$= \{\mathsf{tcstms}\}$
INH(STM)	$= \{\mathsf{Tenv}_{\mathsf{STM}}\}$	SYN(STM)	$= \{\mathsf{tcstm}\}$

constructors and semantic rules:

$$\mathsf{program} : \mathsf{DECLS} \times \mathsf{STMS} \rightarrow \mathsf{PROGRAM} \begin{cases} \mathsf{tcp} = \mathsf{tcstms} \\ \mathsf{Tenv}_{\mathsf{DECLS}} = \mathsf{empty\text{-}env} \\ \mathsf{Tenv}_{\mathsf{STMS}} = \mathsf{tcdecls} \end{cases}$$

$$\mathsf{empty\text{-}decls} : \rightarrow \mathsf{DECLS} \quad \left\{ \mathsf{tcdecls} = \mathsf{Tenv}_{\mathsf{DECLS}} \right.$$

$$\mathsf{decls} : \mathsf{DECL} \times \mathsf{DECLS}_2 \rightarrow \mathsf{DECLS} \begin{cases} \mathsf{tcdecls} = \mathsf{tcdecls}_2 \\ \mathsf{Tenv}_{\mathsf{DECL}} = \mathsf{Tenv}_{\mathsf{DECLS}} \\ \mathsf{Tenv}_{\mathsf{DECLS}_2} = \mathsf{tcdecl} \end{cases}$$

$$\mathsf{stms} : \mathsf{STM} \times \mathsf{STMS}_2 \rightarrow \mathsf{STMS} \begin{cases} \mathsf{tcstms} = \mathsf{and}(\mathsf{tcstm}, \mathsf{tcstms}_2) \\ \mathsf{Tenv}_{\mathsf{STM}} = \mathsf{Tenv}_{\mathsf{STMS}} \\ \mathsf{Tenv}_{\mathsf{STMS}_2} = \mathsf{Tenv}_{\mathsf{STMS}} \end{cases}$$

Figure 2.2: Part of a type checker in an attribute grammar formalism

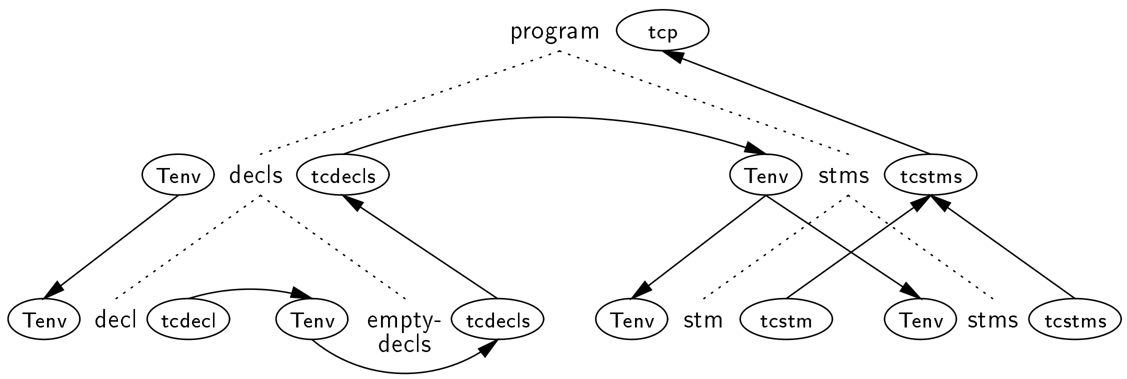


Figure 2.3: Part of an attributed tree of a program

constructors program, empty-decls, decls and stms. We assume that the synthesized attributes tcp, tcstms and tcstm are of type Boolean and that all inherited attributes as well as tcdecls and tcdecl are type-environments. The interpretation I has been omitted: it would provide the semantics of Boolean functions and type-environments.

The attribute rules describe the type checking of a program and its substructures. For the constructor program it states that the type check value, tcp, of a program equals the type check value of the statements, tcstms. The value of the inherited attribute of the declarations in this constructor, $\mathsf{Tenv}_{\mathsf{DECLS}}$, is the constant describing the empty type-environment. The value of the inherited attribute for the statements, $\mathsf{Tenv}_{\mathsf{STMS}}$, equals the type check value of the declarations, tcdecls. □

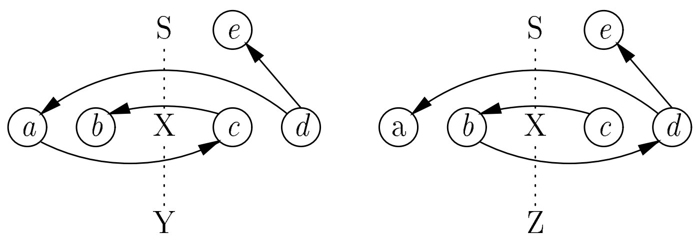
Attributes can be thought of as labels attached to nodes in the abstract syntax tree and their definition rules as a mechanism for transferring information through the tree. Roughly speaking, inherited attributes are used for transferring information down the tree, while synthesized attributes are used for transferring information up the tree.

Example 2.3 Figure 2.3 shows part of the abstract syntax tree of a program, decorated with attributes. Synthesized attributes are displayed on the right-hand side of a tree node, inherited attributes are on the left. The attributes are connected by a *dependency graph* as explained below. □

An attribute pair (a, b) belongs to the attribute dependencies D_p of a constructor $p : X_1 \times \ldots \times X_n \to X_0$ if and only if a is used in the definition of b in the attribute definition rules R_p. To obtain a dependency graph D_T of the attributes of a tree T we take the union of all instances of D_p of constructors p in the tree.

Incremental attribute evaluation

Many algorithms for evaluating attribute values in a tree exist and many of them are incremental. The basis of incremental attribute updating is the attribute dependency graph of an abstract syntax tree. The tree represents, for instance, a program in an editor. When the program is edited, a subtree is replaced in the attributed tree.



$$\text{p: S} \rightarrow \text{X} \begin{cases} b = c \\ a = d \\ e = d \end{cases}$$
$$\text{q: X} \rightarrow \text{Y} \left\{ \; b = a \right.$$
$$\text{r: X} \rightarrow \text{Z} \left\{ \; d = b \right.$$

Figure 2.4: A non-circular attribute grammar which is not strongly non-circular

Attributes of the new subtree must be re-evaluated and also attributes whose values depend directly or indirectly on them. The value of attributes that are not affected by the subtree replacement can be re-used. For instance, when a statement in Figure 2.3 changes, attributes in the declarations and in other statements are not affected.

An attribute grammar is *well-formed* or *non-circular* if for each tree the attribute dependency graph D_T is cycle-free. 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 Figure 2.4 part of an attribute grammar is shown that is not strongly non-circular. The evaluation order of the attributes a,b,c and d at X is determined by the constructor applied at X.

An overview of attribute grammars is given in by Deransart, Jourdan and Lorho in [DJL88]. Attribute grammars are widely used for defining the static semantics of programming languages, e.g, in the Cornell Synthesizer Generator [RT89a, RT89b], the FNC-2 system [JP88], the GAG-system [Kas84], the TOOLS system [KP90] and the Mjølner/ORM Environment [MBD+90].

2.3 Primitive recursive schemes with parameters

Courcelle and Franchi-Zannettacci [CFZ82] have defined a subclass of algebraic specifications, well-presented primitive recursive schemes with parameters (PRSs), that are equivalent to strongly non-circular attribute grammars. We will use this equivalence to implement incremental rewriting for PRSs.

We first give the definition of the class of primitive recursive schemes as described by Courcelle and Franchi-Zannettacci. In Section 2.3.2 we show the basics of the construction of an attribute grammar from a PRS. This construction is valid only if the PRS is well-presented. In Section 2.3.3 we explain why we need well-presentedness and we define this notion.

2.3.1 Definition

In the definition below we define a primitive recursive scheme as a 5-tuple $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$. Intuitively, G is a grammar which consists of a set of free constructor functions, describing, for instance, the abstract syntax of a programming language. Φ is a set of functions over this abstract syntax, like functions for type checking and compilation of programs. The equations Eq_Φ specify the behavior of these Φ-functions. The specification formed by the remaining signature and equations $\langle S, Eq\rangle$ specifies auxiliary data types for the Φ-functions.

Definition 2.3 (PRS) An algebraic specification is a primitive recursive scheme with parameters if it has the following properties

(i) It can be described by a 5-tuple $\langle G, S, \Phi, Eq, Eq_\Phi\rangle$ such that the signature of the specification is formed by the union of G,S and Φ, and the equations of the specification are the union $Eq \cup Eq_\Phi$. G and S are two disjoint signatures, and Φ is a set of functions. Equations over terms of G do not exist in $Eq \cup Eq_\Phi$.

For functions in Φ properties (ii) and (iii) hold.

(ii) The type of the first argument of each ϕ in Φ is a sort in 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 indicates the set of all functions of Φ that have the sort X as the type of their first argument.

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

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{2.1}$$

where τ is a $S \cup \Phi \cup \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$-term.

(a) All x_i and y_j in the left-hand side of equation (2.1) are different variables. Hence, this equation is left-linear.

(b) In equation (2.1) the first argument of any Φ-subterm in τ is an $x_i, 1 \leq i \leq n$ Hence, this equation is strictly decreasing in G.

□

We will use *G-term* to refer to a term of a sort X $\in G$, and *S-term* for a term of sort S $\in S$. On the other hand, we use Φ*-term* to indicate a term of which the head symbol is a function of Φ, and *parameter-term* for the j-th subterm of a Φ-term, $j > 1$. Note that all Φ-terms and parameter terms are S-terms.

The k-th parameter of a function $\phi \in \Phi$ is indicated as $par(\phi, k)$.

Example 2.4 The specification in Figure 2.1 is a part of a PRS. The functions program, empty-decls, decls and stms are abstract tree constructors in G. Φ is the set of type check functions {tcp,tcdecls,tcdecl,tcstms,tcstm}. Equations [Tc1]–[Tc4] are Φ-defining equations. $\langle S, Eq\rangle$ is formed by the specifications of the Booleans and the Type-environments. They are not shown in the figure. □

2.3.2 PRS $\rightarrow$ attribute grammar

In the translation from a PRS $\langle G, S, \Phi, Eq, Eq_\Phi\rangle$ to an attribute grammar $\Gamma = \langle G, ATT, R, I\rangle$, the signature G of the PRS will be identified with the signature G of Γ. The set of attributes ATT will be derived from the set of functions Φ and the attribute definition rules R will be derived from the Φ-defining equations Eq_Φ. Finally, $\langle S, Eq\rangle$ is translated into I, the domain of the attribute values.

When constructing the set of attributes we construct SYN(X) for each X in G by adding a synthesized attribute for each function ϕ in Φ_X. INH(X) is constructed by associating an inherited attribute with each parameter $par(\phi, k)$ of each function ϕ in Φ_X. In some cases one inherited attribute is associated with different parameters. We discuss this in Section 2.3.3.

Example 2.5 We translate the specification in Figure 2.1 into the attribute grammar of Figure 2.2. Synthesized attributes are created for each type check function. Inherited attributes $\mathsf{Tenv}_{\mathsf{DECLS}}$, $\mathsf{Tenv}_{\mathsf{DECL}}$, $\mathsf{Tenv}_{\mathsf{STMS}}$ and $\mathsf{Tenv}_{\mathsf{STM}}$ are created for the parameters of tcdecls, tcdecl, tcstms and tcstm respectively. □

For each constructor $p : X_1 \times \ldots \times X_n \rightarrow X_0$ in G, each Φ-defining equation $eq_{\phi,p}$ yields attribute definition rules R_p for the synthesized attribute ϕ at X_0 and the inherited attributes of children $X_1 \ldots X_n$.

To find the rule for the *synthesized* attribute ϕ at the top node X_0 of constructor p, we replace in a *top-down* traversal of the right-hand side of $eq_{\phi,p}$ all Φ-terms by their associated synthesized attribute, and replace the variables that occur as parameters in the left-hand side by the inherited attributes associated with these parameters.

Example 2.6 From equations [Tc1]–[Tc4] we derive attribution rules for the synthesized attributes tcp, tcdecls and tcstms.

[Tc1] tcp(program(*Decls*,*Stms*)) = tcstms(*Stms*,tcdecls(*Decls*,empty-env))
[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*))

From equation [Tc1] we derive the rule for the, synthesized, tcp attribute of PROGRAM in the constructor program: tcp = tcstms.

Equation [Tc2] yields the rule tcdecls = $\mathsf{Tenv}_{\mathsf{DECLS}}$, with $\mathsf{Tenv}_{\mathsf{DECLS}}$ the inherited attribute for the parameter of tcdecls.

From equation [Tc3] we derive that tcdecls = $\mathsf{tcdecls}_2$, meaning that the tcdecls attribute of DECLS equals the tcdecls attribute of its second child.

Finally, equation [Tc4] yields tcstms = and(tcstm, tcstms_2). □

The attribution rules for *inherited* attributes are derived from the parameter terms in the right-hand sides of Φ-defining equations. In the subterm $\psi(x_i, v_1, \ldots, v_l, \ldots, v_k)$, of the right-hand side of a defining equation $eq_{\phi,p}$, v_l is the parameter term for the l-th parameter of ψ, at the i-th child of constructor p. The procedure to find the attribution rule for the inherited attribute associated with this parameter is similar to the one for synthesized attributes: Replace in a *top-down* traversal of v_l, all Φ-terms by their associated synthesized attributes, and replace the variables that occur as parameter terms in the left-hand side by the inherited attributes associated with these parameters.

Example 2.7 The first parameter of the function tcstms is its second argument, TENV. The associated inherited attribute of STMS is indicated as $\mathsf{Tenv}_{\mathsf{STMS}}$. The inherited attribute associated with the first parameter of tcdecls is indicated as $\mathsf{Tenv}_{\mathsf{DECLS}}$. From the right-hand side of equation [Tc1] we derive the attribution rules for these inherited attributes in constructor program: $\mathsf{Tenv}_{\mathsf{STMS}}$ = tcdecls and $\mathsf{Tenv}_{\mathsf{DECLS}}$ = empty-env. □

2.3.3 Well-presentedness

When translating a PRS into an attribute grammar, it is important that a unique rule is created for each attribute. The attribution rule for a *synthesized* attribute of a constructor is always uniquely determined by the right-hand side of the one defining equation for the associated Φ-function, cf. (iii) in Definition 2.3. For inherited attributes, however, different attribution rules can be derived from the parameters terms in the right-hand sides of Φ-defining equations. We have to define an additional property for primitive recursive schemes which guarantees that in the corresponding attribute grammar inherited attributes are uniquely defined. This property is called *well-presentedness*. The example below shows part of a PRS that is *not* well-presented.

Example 2.8 Equations (2.2) and (2.3) below apply to the same constructor $p : X_1 \times X_2 \rightarrow X_0$.

$$\phi(p(x_1, x_2), y) \quad = \quad \chi(x_1, f(y)) \tag{2.2}$$
$$\psi(p(x_1, x_2), z) \quad = \quad \chi(x_1, g) \tag{2.3}$$

Let inh(ϕ,1), inh(ψ,1) and inh(χ,1) be the inherited attributes associated with the first parameter (the second argument) of ϕ, ψ and χ. The equations yield two different rules for inh(χ,1) at the first child of constructor p, namely: $\mathsf{inh}(\chi, 1) = f(\mathsf{inh}(\phi, 1))$ and $\mathsf{inh}(\chi, 1) = g$. □

The definition of well-presentedness given below, is such that for a well-presented PRS a one-to-one mapping exists between inherited attributes and *variable names* for parameter terms in the left-hand side of defining equations. According to property (iv) in Definition 2.4 each parameter term is represented by exactly one variable. Under this condition an inherited attribute associated with a parameter has a unique definition if all the terms for that same parameter in the right-hand sides of defining equations are *identical* (v). Property (vi) states that two different parameters with identical definitions must be represented by the same variable in the left-hand side as well. The following example illustrates the case in which two parameter terms are represented by one variable, hence are associated with one inherited attribute.

Example 2.9 Consider a PRS with constructors $p : X_1 \to X_0$, $q : X_2 \to X_1$ and equations

$$\phi(p(x_1), y) = f(\psi(x_1, y), \chi(x_1, y)) \tag{2.4}$$
$$\psi(q(x_2), z) = \xi(x_2, z) \tag{2.5}$$
$$\chi(q(x_2), z) = \xi(x_2, z) \tag{2.6}$$

Equation 2.4 defines the parameters $par(\psi, 1)$ and $par(\chi, 1)$ to be equal for the first child of constructor p. Therefore, we must use one variable, z, to represent both parameters in the left-hand sides of the other equations. As a consequence, the two occurrences in equations 2.5 and 2.6, of the parameter term for ξ at the first child of constructor q, are identical. □

Definition 2.4 (Well-presented) A primitive recursive scheme with parameters is *well-presented* if it has the following properties.

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

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{2.7}$$
$$\phi(q(u_1, \ldots, u_{n'}), z_1, \ldots, z_m) = \tau' \tag{2.8}$$

the parameters y_j and z_j are identical for $1 \leq j \leq m$

(v) In right-hand sides of defining equations $\{eq_{\phi,p} | \phi \in \Phi_{X_0}\}$ for the same abstract tree constructor $p : X_1 \times \ldots \times X_n \to X_0$, all subterms with the same Φ-function and over the same X_i

$$\ldots \psi(x_i, v_1, \ldots, v_k) \ldots \qquad \ldots \psi(x_i, v'_1, \ldots, v'_k) \ldots$$

are identical. Hence parameters v_j and v'_j are identical for $1 \leq j \leq k$

(vi) Two *different* parameters, say the j-th parameter of ψ, denoted $par(\psi, j)$, and the l-th parameter, $par(\xi, l)$, of ξ, are represented by the same variable in left-hand sides of defining equations, if and only if for each constructor $p : X_1, \ldots, X_n \to X_0$ it holds that in right-hand sides of defining equations $\{eq_{\phi,p} | \phi \in \Phi_{X_0}\}$ the parameter terms v_j and w_l are identical in all occurrences

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

Note that these Φ-terms have the same first argument x_i.

□

Definition 2.5 (Well-presentable) A primitive recursive scheme with parameters is *well-presentable* if it can be made well-presented by a suitable renaming of variables. □

In their paper, Courcelle and Franchi-Zannettacci give an algorithm for deciding whether a PRS is well-presentable. The algorithm transforms a well-presentable PRS into a well-presented one.

In the sequel we assume that every PRS is well-presented.

2.4 Storage for incremental evaluation

Now that we know that a well-presented primitive recursive scheme is equivalent to a strongly non-circular attribute grammar, we can use an attributed tree for storing normal forms during incremental evaluation of terms in a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$.

We assume that G describes the abstract syntax of some programming language, and that a syntax directed editor for G exists. If a text in this editor is a G-term (a program) an abstract syntax tree T of this text is maintained. When a function $\phi \in \Phi$ is applied to the program in the editor (ϕ could be a type check function or a compilation function) the term $\phi(T)$ is reduced. We want to store normal forms of Φ-terms and parameter terms that occur in this reduction in attributes of the abstract syntax tree T.

In this section we describe the kind of attributes and attribute dependencies that we need for storing information. In Section 2.5 we describe incremental reduction and the updating of the attributes in the stored tree.

2.4.1 Attributes

Attributes in a tree have a name, a value and a status. The name is derived from the Φ-function they belong to. We usually take the function name ϕ as the name of a synthesized attribute. An inherited attribute is named $\mathsf{inh}(\phi,\mathsf{k})$ when it is associated with the k-th parameter $par(\phi, k)$ of ϕ [1].

If an attribute has a non-empty value this value is always a normal form.

The status of an attribute indicates if the attribute contains a correct value or not. If it does not, the status is "Initial" (IN) or "TobeEvaluated" (TE). Otherwise it either has status "Unchanged" (UC) or "Changed" (C). For efficiency reasons that are explained in Section 2.5.2 we need four different status indications rather than just "unreliable" and "reliable".

2.4.2 Attribute dependencies

Attribute dependencies for a PRS are equal to the ones derived from the equivalent attribute grammar. We derive attribute dependencies directly from Φ-defining equations rather than constructing the attribution rules first, as in Section 2.3.2, and then

[1] If one inherited attribute is associated with several parameters like $par(\psi, 1)$ and $par(\chi, 1)$, in equations (2.4)–(2.6) in example 2.9, then its name is a list, e.g. $(\mathsf{inh}(\psi, 1), \mathsf{inh}(\chi, 1))$.

MAKEDEP-EQ($eq_{\phi,p}$)
let $eq_{\phi,p}$ = a defining equation
    τ = the right-hand side of $eq_{\phi,p}$
    ϕ = the synthesized attribute associated function ϕ at X_0
    $D_{\phi,p}$ = a set of dependencies
in
    $D_{\phi,p} :=$ MAKEDEP($\tau, \phi, eq_{\phi,p}$)
    return $D_{\phi,p}$
ni

MAKEDEP(*term*,*att*,$eq_{\phi,p}$)
let $eq_{\phi,p} = \phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$
    term = a subterm of τ
    att = an attribute in the constructor $p: X_1, \ldots, X_n \rightarrow X_0$,
        either the synthesized attribute ϕ of X_0,
        or an inherited attribute of X_i $(1 \leq i \leq n)$
    $D_{\phi,p}$ = a set of dependencies
in
    case
    - *term* $= \psi(x_i, v_1, \ldots, v_k)$, $\psi \in \Phi$
      let ψ_i be the synthesized attribute of function ψ at X_i **in**
        add (ψ_i, att) to $D_{\phi,p}$
        # make dependencies for inherited attributes
        for $j = 1$ **to** k
        do
            let *inh(ψ,j)* be the attribute of the *j*-th parameter of ψ at X_i
            in
                $D_{\phi,p} := D_{\phi,p} \cup$ MAKEDEP(v_j, *inh(ψ,j)*, $eq_{\phi,p}$)
            ni od
      ni
    - *term* $= y_j$, $j \in \{1, \ldots, m\}$
      let *inh(ϕ,j)* be the attribute of the *j*-th parameter of ϕ at X_0 **in**
          add (*inh(ϕ,j)*,*att*) to $D_{\phi,p}$
      ni
    - *term* = f$(s_1, \ldots, s_k)$, f $\notin \Phi$
      for $j = 1$ **to** k **do**
          $D_{\phi,p} := D_{\phi,p} \cup$ MAKEDEP($s_j, att, eq_{\phi,p}$)
      od
    esac
ni
return $D_{\phi,p}$

Figure 2.5: Algorithms for deducing a dependency graph from an equation

deriving the attribute dependencies. Figure 2.5 presents the algorithms for deriving attribute dependencies $D_{\phi,p}$ from an equation $eq_{\phi,p}$. If (att_1, att_2) is a pair in $D_{\phi,p}$ then att_1 is a *predecessor* of att_2 and att_2 is a *successor* of att_1.

MAKE-DEP computes from a term *term* in the right hand side of an equation $eq_{\phi,p}$, the predecessors for the synthesized attribute ϕ at the top of p. For all Φ-terms $\psi(x_i, \ldots)$ in *term*, that are *not* subterms of another Φ-term, a dependency (ψ_i, ϕ) is added to $D_{\phi,p}$. For all y_k that are not subterms of a Φ-term, a dependency $(\mathsf{inh}(\phi, \mathsf{k}), \phi)$ is added to $D_{\phi,p}$. In a similar fashion it derives predecessors of inherited attributes of each subtree x_i of p from parameter terms v_l of $\psi(x_i, v_1, \ldots, v_k)$.

In addition to dependencies derived from equations, we use transitive dependencies between inherited attributes and their *corresponding* synthesized attributes. The corresponding attribute of $\mathsf{inh}(\phi,\mathsf{k})$ is the synthesized attribute ϕ at the same tree node[2]. These dependencies reflect that the normal form of a Φ-term depends on the value of its parameter terms. We will use them as short cuts in the dependency graph during status updating of attributes.

An immediate consequence of the equivalence proved by Courcelle and Franchi-Zannettacci is that the dependency graph D_T is non-circular for all trees T. In their proof they assume a dependency path from each inherited attribute to its corresponding synthesized attributes. Hence, adding the transitive dependencies between corresponding attributes does not change the non-circularity property.

Example 2.10 Figure 2.3 shows the top of the attributed tree we can use when some term tcp(Program) is reduced. In Figure 2.9 the same attributed tree is depicted, but it indicates dependencies between corresponding attributes as well. Moreover, we omitted the names of the attributes and indicated the status instead. A further explanation of this figure will follow in Section 2.5.2. □

In an attributed tree T we distinguish *upward*, *horizontal* and *downward* dependencies. *Upward* dependencies run from a synthesized attribute of a subtree T' of T to a synthesized at the parent of T'. *Horizontal* dependencies run from a synthesized attribute of a subtree T' to an inherited attribute of a sibling of T' or of T' itself. Also the dependency between an inherited attribute and a corresponding synthesized attribute is horizontal. *Downward* dependencies run from an inherited attribute to an inherited attribute at a lower node.

2.5 Incremental evaluation

Assume T is the attributed abstract syntax tree of a program in an editor. During incremental reduction of a Φ-term $\phi(T)$ we store the normal forms of Φ-terms and parameter terms in attributes of T. After this the text in the editor can be edited again. As a result of editing subtrees are replaced in T, and attributes that have become unreliable are marked. Let us say that the new abstract syntax tree is T'.

[2] If the name of an inherited attribute is a list $(\mathsf{inh}(\psi, 1), \mathsf{inh}(\chi, 1))$ then its corresponding attributes are ψ and χ.

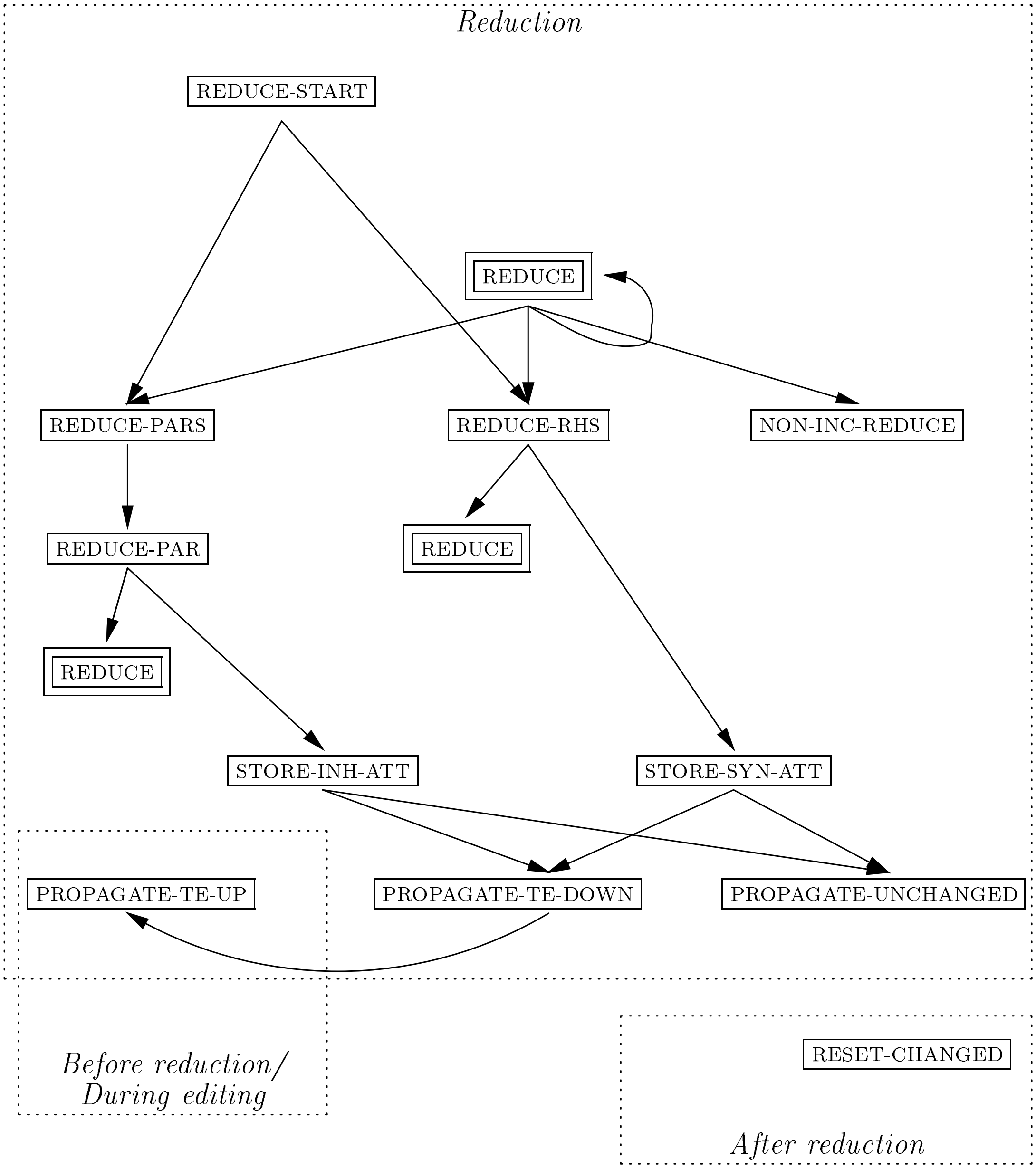


Figure 2.6: Overview of algorithms for incremental rewriting

When subsequently the user wants the know the ϕ value of the modified program, the term $\phi(T')$ is to be reduced. During reduction of this term, reduction of terms of which the associated attribute value is unaffected is not necessary, since the term can be replaced by the stored attribute value. Also the status of attributes is updated during reduction

Figure 2.6 presents an overview of the algorithms we use for incremental rewriting. The REDUCE algorithms specify incremental reduction of a term $\phi(T)$ in the presence of an abstract syntax tree T with an attribute dependency graph D_T. They make use of algorithms for storing a normal form in an attribute. These algorithms in turn make use of propagation algorithms for updating the status of the successors of the attribute in the dependency graph D_T.

In Section 2.5.1 we first describe incremental reduction as specified by the REDUCE algorithms. In Section 2.5.2 we describe the updating of the attributes in the stored tree at different stages: before reduction (during editing) when PROPAGATE-TE-UP is applied, during reduction as specified by the STORE algorithms and the PROPAGATE algorithms, and after reduction, when the algorithm RESET-CHANGED is applied.

2.5.1 Reduction

The reduction strategy we use is the *leftmost innermost strategy*, that is, when a term is to be reduced we first reduce all its subterms recursively, starting with the leftmost one.

The algorithms in Figure 2.7 and Figure 2.8 describe incremental reduction. We first explain the reduction steps, then we explain how we keep track of the proper subtree in the attributed tree.

When REDUCE-START and REDUCE are applied to a Φ-term the associated synthesized attribute is inspected first. If the attribute has a correct value, that is, the status of this attribute is not "TobeEvaluated" and not "Initial", we can skip further reduction and replace the term by the normal form in the attribute value. Otherwise we have to reduce the term. Since we use innermost reduction the parameter terms are reduced first by applying REDUCE-PARS, the result is then reduced by REDUCE-RHS.

When a parameter term is to be reduced (REDUCE-PARS, REDUCE-PAR), its associated inherited attribute is inspected. If it has a correct value the parameter term is replaced by the attribute value. Otherwise, the term is reduced and the resulting normal form is stored in the attribute.

After this, in REDUCE-RHS, we check again the status of the synthesized attribute, as it may occur that none of the values of the inherited attributes have changed and that as a consequence the synthesized attribute need not be re-evaluated (Section 2.5.2). If the attribute status is still "TobeEvaluated" or "Initial", the term is reduced further by selecting the one equation with a matching left-hand side, instantiating its right-hand side, and REDUCE this term. The result is stored in the synthesized attribute.

When REDUCE is applied to a non-incremental term, the subterms are reduced first by a recursive call to REDUCE. Then the term is reduced in a standard, non-incremental fashion by NON-INC-REDUCE.

```
# In all algorithms T is an abstract syntax tree
# T is decorated with an attribute dependency graph D_T.
# Equations are considered global information.

REDUCE-START(topterm,T)
let topterm = φ(p(S_1,...,S_n),t_1,...,t_m)
    s.t. p(S_1,...,S_n) equals the attributed term T
    φ = synthesized attribute for function φ at the root of T
    reduced-topterm = a term
in
    if status_φ ≠ "TobeEvaluated" or "Initial"
    then reduced-topterm := value_φ
    else reduced-topterm := REDUCE-PARS(topterm,T,T)
         reduced-topterm := REDUCE-RHS(reduced-topterm,T,φ)
    fi
    return reduced-topterm
ni

REDUCE(term,T)
let T_i = the i-th subtree of T
    S_i = a term p(S'_1,...,S'_k) which equals T_i
    reduced-term = a term
in case
    • term = φ(S_i,v_1,...,v_k), φ ∈ Φ
        let φ_i be the synthesized attribute for function φ at T_i
        in if status_φi ≠ "TobeEvaluated" or "Initial"
           then reduced-term := value_φi
           else reduced-term := REDUCE-PARS(term,T,T_i)
                reduced-term := REDUCE-RHS(reduced-term,T_i,φ_i)
           fi ni
    • term = f(s_1,...,s_k), f ∉ Φ
        for j = 1 to k do
            reduced-s_j := REDUCE(s_j,T) od
        reduced-term := NON-INC-REDUCE(f(reduced-s_1,...,reduced-s_k))
   esac
   return reduced-term
ni

REDUCE-RHS(term,T,φ)
let reduced-term = a term, inst-τ = a term
in
    if status_φ ≠ "TobeEvaluated" or "Initial"
    then reduced-term := value_φ
    else term matches equation eq_φ,p with right hand side τ
         inst-τ := instantiated τ
         reduced-term := REDUCE(inst-τ,T)
         STORE-SYN-ATT(reduced-term, φ)
    fi
    return reduced-term
ni
```

Figure 2.7: Algorithms for incremental reduction of a Φ-term

```
# In all algorithms T is an abstract syntax tree
# T is decorated with an attribute dependency graph D_T.
# T_i is the i-th subtree of T,

REDUCE-PARS(term,T,T_i)
let term = φ(S_i, t_1, ..., t_m)  s.t. S_i equals T_i
    reduced-term, reduced-t_j = terms
in  for j = 1 to m
      do
        let inh(φ,j) be the attribute for the j-th parameter of φ at T_i
        in
            reduced-t_j := REDUCE-PAR(t_j,inh(φ,j),T)
        ni
      od
      reduced-term := φ(T_i, reduced-t_1, ..., reduced-t_m)
      return reduced-term
ni

REDUCE-PAR(parameter-term,att,T)
let parameter-term, reduced-parameter = terms
    att is an inherited attribute.
in
    if status_att ≠ "TobeEvaluated" or "Initial"
    then
       reduced-parameter-term := value_att
    else
       reduced-parameter-term := REDUCE(parameter-term,T)
       STORE-INH-ATT(reduced-parameter-term,att)
    fi
    return reduced-parameter-term
ni
```

Figure 2.8: Algorithms for incremental reduction of a parameter term

Keeping track of the subtrees in the attributed tree

In all reduce algorithms we keep track of subtrees T and T_i in the attributed tree. Thus we know where to look for associated attributes. Initially, in REDUCE-START, the attributed tree equals the first subterm of the Φ-term to be reduced. When we proceed with the reduction of the right-hand side of the equation that matches the Φ-term, we keep track of this same tree: REDUCE(τ, T). The "strictly decreasing" property of defining equations guarantees that the first subterm of each Φ-term in τ equals a subtree T_i of T. In REDUCE-PAR we need a reference to the subtree T_i of T, that equals the first subterm of the Φ-term, as well as a reference to T. This is necessary for further reduction of parameter terms.

Example 2.11 When the term tcp(program(Decls,Stms)) is being reduced, it matches the left-hand side of [Tc1]. REDUCE proceeds with the term tcstms(Stms,tcdecls(Decls,empty-env)) and a reference to the attributed tree Program. REDUCE-PARS is applied with references to Program and its subtree Stms. Stms is needed because it has the attributes for the parameters of tcstms. REDUCE-PAR applies REDUCE with the parameter term tcdecls(Decls,empty-env) and the pointer to Program, so that we can find the subtree Decls for further reduction. □

2.5.2 Updating the attributed tree

Crucial in incremental evaluation is the status of the attributes. Figure 2.6 presents the relation between the REDUCE algorithms, the STORE algorithms for storing values in attributes, and the PROPAGATE algorithms for resetting the status of attributes. In this section we explain in more detail when and how the status of attributes is updated.

Initially, before a function $\phi \in \Phi$ has been applied to an edit term T attributes in T have an empty value and status "Initial". During reduction of $\phi(T)$, the attribute ϕ of T as well as all its predecessors in the attribute dependency graph of T obtain a (new) value. Their status is then set to "Changed". After reduction all "Changed" attributes are reset to "Unchanged".

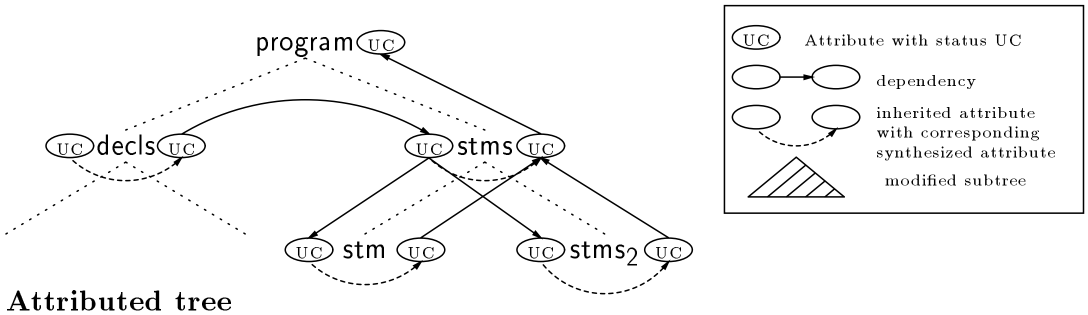
Example 2.12 The first tree in Figure 2.9 shows the status of the attributes in a tree after initial reduction of some term tcp(Program). □

After this initial reduction the storage is updated at four different stages: during editing, right before reduction, during reduction when an attributed gets a new value, and after reduction to reset the status of all "Changed" attributes to "Unchanged".

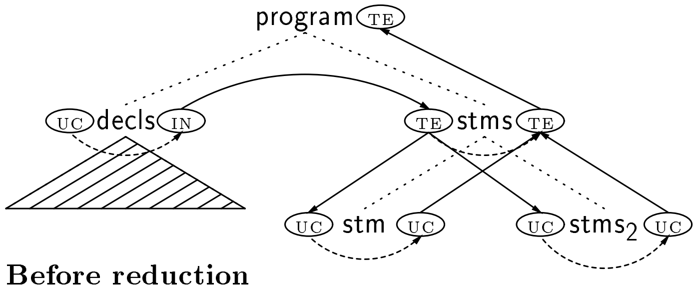
Status indications

We first give a description of the differences between the several status indications.

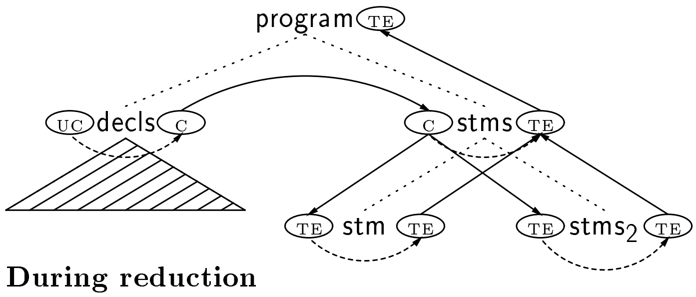
Both status "Changed" and status "Unchanged" indicate that an attribute value is reliable. During reduction, attributes obtain status "Changed" when they have



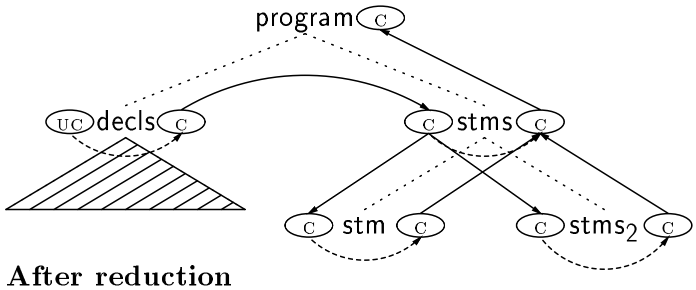
Attributed tree



Before reduction
PROPAGATE-TE-UP has been applied to the synthesized attribute at decls



During reduction
PROPAGATE-TE-DOWN has been applied to the inherited attribute at stms



After reduction
before RESET-CHANGED is applied

Figure 2.9: Attributed tree at various stages of incremental rewriting

obtained a new (correct) value. Status "Unchanged" is used for attributes that have a correct value that has not been changed during reduction.

We use status "Initial" and "TobeEvaluated" for attributes of which the value is not necessarily correct in the attributed tree. The difference between the two is that the value of a "TobeEvaluated" attribute is consistent with the value of its predecessors with status "Unchanged", "TobeEvaluated" and "Initial". This need not be the case for "Initial" attributes. When during reduction the values of the predecessors of a "TobeEvaluated" attribute remain the same, the "TobeEvaluated" attribute need not be re-evaluated and its status can be reset to "Unchanged". In contrast, the value of an "Initial" attribute must always be re-computed.

Attribute updating during editing; multiple subtree replacements

Assume that T is an attributed abstract syntax tree of a program in an editor and that attributes in T have obtained a value when ϕ or any other Φ-function was applied to T.

After a modification in a part of the text in the editor the modified part is re-parsed and a subtree *Oldsub* in T is replaced by *Newsub*. *Newsub* is decorated with attributes with an empty value and status "Initial". The top node of *Newsub* gets the attributes of the top node of *Oldsub*. The synthesized attributes at the top still contain the normal forms of terms $\chi(\mathit{Oldsub}, \ldots)$, $\chi \in \Phi$. They are no longer reliable and their value is not consistent with the value of their predecessors. Their status is therefore set to "Initial".

As a result of a subtree replacement, the value of attributes that depend on synthesized attributes at the replacement node (*affected attributes*) have become unreliable. We do not mark all affected attributes as unreliable before reduction starts. It suffices to do so only for those that are on a path of horizontal and upward dependencies from a synthesized attribute of a replacement node to an attribute at the top, because other affected attributes can be marked during reduction. After the algorithm PROPAGATE-TE-UP in Figure 2.11 has been applied to a synthesized attribute at the top of *Newsub*, all its *horizontal* and *upward* successors have status "TobeEvaluated" or "Initial".

Example 2.13 The second tree in Figure 2.9 shows the effect of updating an attributed tree in which the declarations have been replaced. □

Editing may proceed and other subtrees will then be replaced in the abstract syntax tree. After each subtree replacement the same procedure is applied. Note that PROPAGATE-TE-UP does not continue if the status of an attribute is already "TobeEvaluated" or "Initial".

Attribute updating before reduction

In case the user of the editor wants to apply a function $\psi \in \Phi$ to the program T', and ψ is a function with parameters, then some term $\psi(T', t'_1, \ldots, t'_m)$ is to be reduced. The parameter terms t'_j can differ from the values of the inherited attributes $\mathsf{inh}(\psi, \mathsf{j})$. We

STORE-SYN-ATT($term$,ϕ)
let $term$ = the normal form of a Φ-term
    ϕ is a synthesized attribute
in
  if $value_{\phi} \neq term$
  then $value_{\phi} := term$
      $status_{\phi}$:= "Changed"
  else $status_{\phi}$:= "Unchanged"
      PROPAGATE-UNCHANGED(ϕ)
  fi
ni

STORE-INH-ATT($term$,att)
let $term$ = the normal form of a parameter term
    att is an inherited attribute
in
  if $value_{att} \neq term$
  then $value_{att} := term$
      $status_{att}$:= "Changed"
      PROPAGATE-TE-DOWN(att)
  else $status_{att}$:= "Unchanged"
      PROPAGATE-UNCHANGED(att)
  fi
ni

Figure 2.10: Algorithms for storing normal forms in attributes and propagate information to successor attributes

therefore mark the associated inherited attributes of the parameters of ψ as "Initial", and apply PROPAGATE-TE-UP from there on. As a result, the synthesized attribute ψ of T either has status "Initial" or status "TobeEvaluated".

Attribute updating during reduction

The term $\phi(T')$ (or $\psi(T', t'_1, \ldots, t'_m)$) is reduced incrementally in the presence of the attributed tree T' exactly as has been described in the algorithms of Figure 2.7 and 2.8.

The algorithms STORE-SYNATT and STORE-INHATT in Figure 2.10 are applied when a normal form must be stored in a synthesized attribute or an inherited attribute. The new attribute value is compared to the old value, if they are different the status of the attributed is set to "Changed", otherwise it becomes "Unchanged".

A changed inherited attribute of a tree T' can have successor attributes of subtrees of T' (via a *downward* dependency) that are not yet marked as unreliable. After application of PROPAGATE-TE-DOWN all direct and transitive successors at the *top nodes* of the subtrees of T are marked as "TobeEvaluated" or "Initial".

Example 2.14 The third tree in Figure 2.9 shows the effect of applying PROPAGATE-TE-DOWN after the inherited attribute of stms has obtained a new value. □

When an attribute has been re-evaluated, and its value is found not to have changed after all there is a chance that its successors do not have to be re-evaluated. If the status of a successor is "TobeEvaluated" and it has only "Unchanged" predecessors, its status can be reset to "Unchanged" as well. PROPAGATE-UNCHANGED takes care of this.

This algorithm is the reason why we need two status indications for attributes with a reliable value and two for attributes with an unreliable value: Only "TobeEvaluated" attributes are reset to "Unchanged", not "Initial" attributes. Moreover, the status of a "TobeEvaluated" attribute is reset only if all its predecessors has status "Unchanged". Successors of a "Changed" attribute must always be re-evaluated.

Attribute updating after reduction

When the reduction process has ended, RESET-CHANGED is applied to reset the status of all "Changed" attributes to "Unchanged", and the status of their direct "TobeEvaluated" successors as well their corresponding attributes to "Initial".[3]

Example 2.15 The last tree in Figure 2.9 shows the attributed tree after reduction has ended, and before RESET-CHANGED is applied. Replacing declarations is a good example to illustrate the propagation algorithms, on the other hand it is unfortunate that all attributes had to be re-evaluated because of this modification. Note that after a statement replacement more attribute values can be re-used. In Section 2.9 we come back to the consequences of modifying declarations. □

[3]For efficiency reasons, "Changed" attributes must be added to a list during reduction. This avoids a search through the whole tree. We have omitted this aspect from the algorithms.

```
PROPAGATE-TE-UP(φ)
let φ, ψ, successor = attributes in a dependency graph D_T
in  for all direct successor attributes successor of φ
    do  when  status_successor ≠ "TobeEvaluated" or "Initial"
          do
             status_successor := "TobeEvaluated"
             if  successor is a synthesized attribute
             then
                  PROPAGATE-TE-UP(successor)
             else
                  successor is an inherited attribute
                  for  all its corresponding synthesized attributes ψ
                  do   status_ψ := "TobeEvaluated"
                       PROPAGATE-TE-UP(ψ)
                  od
            fi
          od od
ni
```

Figure 2.11: Algorithm for upward propagation of the status "TobeEvaluated", starting at a synthesized attribute.

```
PROPAGATE-TE-DOWN(inhatt)
let inhatt = an inherited attribute
    ψ, successor = attributes
in  for all direct successor attributes successor in D_T of inhatt
    do  when successor is an inherited attribute
          and   status_successor ≠ "TobeEvaluated" or "Initial"
          do
             status_successor := "TobeEvaluated"
             for all corresponding synthesized attributes ψ of successor
             do  status_ψ := "TobeEvaluated"
                  PROPAGATE-TE-UP(ψ)
             od
          od
    od
ni
```

Figure 2.12: Algorithm for downward propagation of status "TobeEvaluated", starting at an inherited attribute.

```
# att is an attribute with status "Unchanged" in a dependency graph D_T

PROPAGATE-UNCHANGED(att)
let att = an attribute with status "Unchanged" in a dependency graph D_T
    successor = an attribute
in
   for all direct successor attributes successor in D_T of att
   do
    when  status_successor is "TobeEvaluated"
    do
       when the status of all direct predecessors  of successor in  D_T is "Unchanged"
       do
           status_successor := "Unchanged"
          PROPAGATE-UNCHANGED(successor)
       od
    od od
ni
```

Figure 2.13: Algorithm for propagating status "Unchanged".

```
RESET-CHANGED(att)
let att = an attribute with status "Changed" in a dependency graph D_T
in
    status_att := "Unchanged"
    for all direct successor attributes in D_T of att
    do
        status_successor := "Initial"
    od
    when att is an inherited attribute
    do
       for all  corresponding synthesized attributes in D_T of att
       do
           status_successor := "Initial"
       od
    od
ni
```

Figure 2.14: Algorithm for resetting the status of a "Changed" attribute.

We explain why RESET-CHANGED also resets the status of the "TobeEvaluated" successors of "Changed" attributes. In our running example no "TobeEvaluated" attributes appear when reduction is finished. If a PRS contains two or more groups of Φ-functions, for instance, functions for type checking and for compilation, and the two groups have some functions in common, then a tree is decorated with a dependency graph for each group of functions, and these graphs have common subgraphs. When one group of functions has been evaluated, the attributes for these functions all have status "Unchanged" or "Changed". Attributes associated with other functions can have status "TobeEvaluated", while having a predecessor (in the common subgraph) with status "Changed". The value of these "TobeEvaluated" attributes is not consistent with the value of its "Changed" predecessor. So when the status of a "Changed" attribute is reset to "Unchanged" the status of these "TobeEvaluated" successors must be changed to "Initial".

2.6 Correctness

A detailed correctness proof of the presented algorithms is given in Appendix A. Here we present an overview of the proof.

As mentioned before, a tree can be decorated with dependency graphs for several groups of functions, and during reduction of a term $\phi(T, t_1, \ldots, T_m)$ only attributes associated with functions that occur in this reduction are re-evaluated. So we cannot prove that after reduction a tree is correctly attributed. Therefore, we introduce the notion of a safely attributed tree, and demonstrate that given a safely attributed tree, T, incremental reduction of $\phi(T, t_1, \ldots, t_m)$ returns a correct normal form and again a safely attributed tree T. Application of RESET-CHANGED also preserves the safety of T. Preprocessing steps in a safely attributed tree preserve safety as well. Finally, before any incremental reduction has taken place, the tree is safely attributed.

The definitions of safe attributes and a safely attributed tree are as follows.

Definition 2.6 Let T be an attributed tree. An attribute Att in T is *safe* if it has one of the following properties.

- Its status is "Initial".
- Its status is "TobeEvaluated", and its value is correct w.r.t its direct predecessors with status "Initial","TobeEvaluated" or "Unchanged".
- Its status is "Unchanged" or "Changed" and its value is correct w.r.t. its direct predecessors, and its value is correct w.r.t. T
- Its status is "Unchanged" or "Changed" and its value is correct w.r.t. its direct predecessors, and it has an (indirect) inherited predecessor with status "Initial" or "TobeEvaluated", at an ancestor node of T_{Att}.

□

Definition 2.7 An attributed tree T is *safely attributed* if each attribute is safe in D_T, and moreover, all upward and horizontal successors of an attribute with status "Initial" or "TobeEvaluated" have either status "Initial" or status "TobeEvaluated". □

2.7 Discussion

The design of our algorithms has been influenced by our wish to be both efficient and to remain as close as possible to a standard rewrite strategy.

The strategy for incremental rewriting can be thought of as leftmost-innermost rewriting with short cuts and side effects.

The choice for an innermost strategy has been fortunate, an outermost strategy would not provide such a clear visiting order for attributes. In an outermost strategy it would be more complicated to keep track of the proper subtree in the attributed tree during reduction, and the propagation of status "TobeEvaluated" would have been more greedy.

Our algorithms can handle multiple subtree replacements. This is relevant because the method is to be implemented as part of the rewrite system of the ASF+SDF Meta-environment [Kli93b]. We wanted it to be such that a user can freely edit a program without being hindered by more or less expensive intermediate rewriting. Evaluation of functions applied to the program, is to take place only on explicit request by the user.

Related to the previous point is that the algorithm has a *data driven* part and a *demand driven* part. In this it resembles the approach for lazy and incremental attribute updating as proposed by Hudson [Hud91]. The data driven part is the status updating performed by PROPAGATE-TE-UP and is called after each edit action that causes a subtree replacement in the (attributed) abstract syntax tree of the text in the editor. Status propagation is expected to be a very fast operation. The re-evaluation of attributes is demand driven, because only attributes that are visited during rewriting of, for instance, a type check term are updated. A tree may be decorated with attributes for type checking and attributes for compilation, and the user only wants to know the new type check value. Attributes for compilation are then not re-evaluated.

The price we pay for remaining close to a standard rewrite strategy is that the algorithm is not *optimal*. An algorithm is called optimal if the number of attributes that are re-evaluated after one ore more subtree replacements is $O(|\mathit{Affected}|)$, with *Affected* the attributes that get a new value during re-evaluation. Let *paths_to_root* be the set of attributes on the dependency path from a new subtree to the top. These attributes are always visited before reduction by PROPAGATE-TE-UP, and during reduction. As a result, the number of re-evaluated attributes is $O(|\mathit{Affected}| + |\mathit{paths\_to\_root}|)$.

A previous approach

Reps, Teitelbaum and Demers have described an optimal time algorithm for updating attribute values in the context of attribute grammars [RTD83]. An earlier approach for incremental rewriting [Meu92] was patterned after this algorithm.

In short: after each edit action, re-evaluation starts at the point of subtree replacement. When a synthesized attribute at the top of the new subtree has corresponding inherited attributes, that could not be affected by the subtree replacement, a term is constructed for the synthesized attribute and (incrementally) rewritten. When the normal form of this term differs from the attribute value, the status is set to "Changed", and successor attributes up to the parent node of the new subtree are marked as "TobeEvaluated". The same procedure is applied to this parent tree. Re-evaluation stops when no "TobeEvaluated" attributes are left.

The advantage of this algorithm is that attributes are re-evaluated if and only if a predecessor has changed value.

Disadvantages are that the algorithm is difficult to adapt to handle multiple subtree replacements and is less efficient in practice because it is completely data driven. The construction of terms makes the algorithm more complicated than the one presented in this chapter.

2.8 Related work

The only other work dealing with incremental rewriting we know of, is a recent article by Field [Fie93]. His method applies to arbitrary left-linear term rewriting systems. In a term T to be rewritten, subterms can be designated as *substituends*. Complementary to the set of substituends in a term T is the set of context terms for all possible combinations of substituends. When T is rewritten all context terms are rewritten simultaneously. When subterms in substituends positions in T are replaced, the normal form of the related context term is used as a starting point for subsequent reduction of the altered term. In order to obtain simultaneous rewriting of a term and the context terms, both the rewrite system itself and the rewrite strategy are adapted. The method relies on graph rewriting to prevent the introduction of extra rewrite steps for reducing the context terms. Whereas this method is elegant and applicable to general rewrite systems, we expect that our method is more efficient for the specific class of specifications we are interested in. It is not well understood how the two methods could be compared in a more formal manner.

There are many papers dealing with the relation between attribute grammars and other formalisms. The results of Courcelle and Franchi-Zannettacci are used in papers by Attali and Franchi-Zannettacci and Jourdan [AFZ88, Att88, Jou84].

Jourdan starts with a strongly non-circular attribute grammar and compiles it to a primitive recursive scheme. One of his objectives is to solve the space problem that usually arises with attribute grammars. 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, CDD$^+$85], and is closely related to PROLOG. The first implementation of Typol was therefore based on PROLOG. 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 translated to strongly non-circular attribute grammars, in either the Synthesizer Specification Language, SSL [RT89a, RT89b] or Olga, the input formalism for the FNC-2 system [JBP90, JP88]. In later papers [AC90, ACG92], Attali and Chazarain use the relation between attribute grammars and Typol programs to determine a class of Typol programs for which functional evaluators based on pattern matching can be generated. An incremental implementation of these evaluators is based on an attribution of the proof tree. In this particular class of Typol specifications a natural relation exists between parts of the abstract syntax tree of the program in the editor and parts of the proof tree. When the program is edited, the corresponding parts of the proof tree are replaced, subsequent evaluation computes the values of the attributes of the new parts of the proof tree and of all attributes that depend on them.

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

Pugh and Teitelbaum discuss in [PT89] how function caching can be used for incremental evaluation of attribute grammars that have been translated according to Katayama's scheme. In the resulting procedure, like in ours, the number of attributes that are re-evaluated after a subtree replacement is $O(|Affected| + |path\_to\_root|)$. This suggests that we could just as well have applied function caching directly to our class of algebraic specifications. However, for efficient implementation of function caching two additional problems have to be solved. Firstly, a way of storing function calls must be found, so that pending calls can be compared with entries in the cache in constant time, even when the function calls have large arguments. Secondly, techniques for purging the cache must be developed, so that it will not be filled up with old and useless information.

In [PSV92, VSK89, Vog93] Vogt et al. develop an incremental method for ordered attribute grammars, a subclass of strongly non-circular attribute grammars, based on function caching. In their approach multiple instances of the same tree are shared by means of memoed tree constructors. In this way, comparing function calls with entries in the cache reduces to simple pointer comparison. However, they do not address the question for a satisfying purging method.

2.9 Conclusions

We developed a technique for incremental rewriting for primitive recursive schemes. It is based on storing normal forms in attributes of a parse tree, and is capable of dealing with multiple subtree replacements.

It is not surprising that the class of well-presented primitive recursive schemes, like attribute grammars, is a very natural one for specifying the static semantics of languages. Many specifications written in ASF+SDF have the flavor of a PRS, and could easily be transformed into one. Examples are the type checking of Asple, Pascal and mini-ML as described in [Meu88, Deu91, Hen89b]. Two principal non-PRS features of ASF+SDF are used in these specifications, namely conditional equations and list sorts [Hen91]. In the next chapter we extend our method to PRSs with conditional equations. In Chapter 4 we describe an incremental implementation of functions over lists.

We already pointed out at the end of Section 2.5.2 that a change in the declarations which results in a new type check value, forces an inefficient re-type check of the complete statement section. In Chapter 5 we remedy this shortcoming by extending the incremental strategy to auxiliary data types.

Chapter 3

Conditional incremental rewriting

In the previous chapter we have presented a technique for deriving incremental implementations for a subclass of algebraic specifications, namely, well-presented primitive recursive schemes. We used 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.

In this chapter the technique is extended to primitive recursive schemes with conditional equations.

The main extension in the algorithm is the dynamic adaptation of the attribute dependency graph of a tree during evaluation. This is done for efficiency reasons, so that useless recomputation, are avoided.

A simple adaptation of the derivation of dependencies from equations is needed in order to deal with the use of new variables in conditions.

3.1 Introduction

In the previous chapter we have described a technique for incremental reduction of terms in a well-presented primitive recursive scheme with parameters (PRSs).

Such a scheme is equivalent to a strongly non-circular attribute grammar. Therefore we are able to store normal forms of functions over a G-term -typically the abstract syntax tree of a program in an editor- in synthesized attributes of that G-term. Parameters of these function applications are stored in inherited attributes. Attributes are connected by means of a dependency graph so that after a modification in the G-term unreliable attribute values can be detected. Upon a subsequent application of functions to a similar G-term, reliable attribute values can be used to skip reduction steps, unreliable attribute values are re-computed.

In this chapter we extend our technique to conditional PRSs. Many algebraic specifications of static semantics of languages meet the requirements of a well-presented primitive recursive scheme. Conditional well-presented primitive recursive schemes, however, provide more flexibility. We could of course transform a conditional specification into one without conditions, and apply the incremental evaluation technique described in the previous chapter to the resulting specification. However, for innermost rewriting strategies, it is known that the resulting non-conditional specification is less efficient than the original one, because it may happen that terms are rewritten that would have been excluded by the conditions, and the result of which does not contribute to the final result.

Therefore, we present a method for conditional incremental rewriting that can be directly applied to conditional primitive recursive schemes. Moreover, we update attribute *dependencies* during reduction by removing dependencies derived from equations that were not used in the reduction.

Conditions may introduce new variables that were not introduced in the left-hand side of an equation. We adapt the derivation of attribute dependencies to handle this.

Although the principle of dynamic updating of attribute dependencies, as described in this chapter, is applicable to incremental implementation of conditional attribute grammars, we have found no account of such work.

Overview of this chapter

In Section 3.2 we list the properties of simple conditional PRSs. In Section 3.3 we discuss incremental evaluation with dynamic updating of the dependency graph. Section 3.4 discusses the restrictions on the use of new variables and the derivation of attribute dependencies. Section 3.5 presents some conclusions.

3.2 Conditional primitive recursive schemes

Conditional primitive recursive schemes constitute a superclass of primitive recursive schemes as defined in Definition 2.3 and provide more flexibility for writing specifications.

In a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$ with conditional equations several defining equations for each pair (ϕ, p) may exist in Eq_Φ.

$$\frac{\lambda_1 = \rho_1, \ldots, \lambda_k = \rho_k}{\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau} \tag{3.1}$$

$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$ is the *conclusion* of the equation, $\lambda_i = \rho_i$ are the conditions.

$EQ_{\phi,p}$ indicates the list of equations with left-hand side $\phi(p(\ldots), \ldots)$; $eq_{\phi,p,j}$ is the *j-th* equation in this list.

Initially, we assume that the restrictions formulated in (ii) and (iii) of Definition 2.3 that apply to the term on the right-hand side τ also apply to the terms appearing in the conditions. Namely, that the terms λ_i and ρ_i are of type S $\in S$, and that they are terms over the signature $S \cup \Phi \cup \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$. Also the well-presentedness property (Definition 2.4) for parameter terms in the right-hand side should apply to λ_i and ρ_i.

In Section 3.4 we lift one restriction by allowing new variables in conditions. This extension does not affect the incremental algorithm discussed in the next section.

Example 3.1 Equations [Evif1] and [Evif2] below describe the dynamic semantics of an if-statement. The evaluation functions evs,eve,evss are Φ-functions. The variable *Env* represents a value-environment, a table with identifiers and their values, which is updated during evaluation. The first argument of the if-constructor is an integer expression that evaluates to 0 or 1.

$$\text{[Evif1]} \quad \frac{\mathsf{eve}(\mathit{Exp}, \mathit{Env}) = 0}{\mathsf{evs}(\mathsf{if}(\mathit{Exp}, \mathit{Stms}_1, \mathit{Stms}_2), \mathit{Env}) = \mathsf{evss}(\mathit{Stms}_1, \mathit{Env})}$$

$$\text{[Evif2]} \quad \frac{\mathsf{eve}(\mathit{Exp}, \mathit{Env}) = 1}{\mathsf{evs}(\mathsf{if}(\mathit{Exp}, \mathit{Stms}_1, \mathit{Stms}_2), \mathit{Env}) = \mathsf{evss}(\mathit{Stms}_2, \mathit{Env})}$$

□

3.3 Incremental evaluation with conditional equations

Incremental rewriting for conditional PRSs is almost the same as for non-conditional PRSs, except that for efficiency reasons dependencies are updated dynamically. We briefly mention the construction of attribute dependencies, we then explain incremental rewriting, and finally we explain the optimization by adding and removing attribute dependencies dynamically.

3.3.1 Attribute dependencies

Assume T is the abstract syntax tree of a text in an editor. If no Φ-function has been applied to T, T is *fully attributed*.

The attribute dependencies for each abstract tree constructor p in T are obtained by taking the union of the dependencies of all equations that apply to that constructor: $D_p = \bigcup_{\phi,j} D_{\phi,p,j}$.

Figure 3.1 presents the algorithm for deriving dependencies $D_{\phi,p,j}$ from the conditions and the right-hand side of the conclusion of equation $eq_{\phi,p,j}$. It uses the algorithm MAKEDEP in Figure 2.5.

MAKEDEP computes from a term *term* in a condition or in the right hand side of an equation $eq_{\phi,p,j}$ the predecessors for the synthesized attribute ϕ at the top of p. For all Φ-subterms $\psi(x_i, \ldots)$ of *term* that are *not* subterms of another Φ-term, a dependency (ψ, ϕ) is added to $D_{\phi,p,j}$. For all y_i that are not subterms of a Φ-term, a dependency $(\mathsf{inh}(\phi, \mathsf{i}), \phi)$ is added to $D_{\phi,p,j}$. In a similar fashion it derives predecessors of inherited attributes of each subtree x_i of p from parameters v_l of $\psi(x_i, v_1, \ldots, v_k)$.

Example 3.2 Figure 3.2 shows dependencies that have been derived from equations [Evif1] and [Evif2] in Example 3.1. The dashed arrows are additional dependencies that connect inherited attributes and their *corresponding* synthesized attributes. □

MAKEDEP is the algorithm for deriving depedencies from a subterm of the right hand side or condition. It is shown in Figure 2.5 on page 23.

MAKEDEP-CONDEQ($eq_{\phi,p,j}$)
let $eq_{\phi,p,j} \;=\; \frac{\lambda_1=\rho_1,\ldots,\lambda_k=\rho_k}{\phi(p(x_1,\ldots,x_n),y_1,\ldots,\;y_m)=\tau}$
  ϕ = the attribute of function ϕ at X_0
  $D_{\phi,p,j}$:= a set of attribute dependencies
in
  for $i = 1$ to k
  do
    $D_{\phi,p,j} := D_{\phi,p,j} \cup \text{MAKEDEP}(\lambda_i, \phi, eq_{\phi,p,j})$
    $D_{\phi,p,j} := D_{\phi,p,j} \cup \text{MAKEDEP}(\rho_i, \phi, eq_{\phi,p,j})$
  od
  $D_{\phi,p,j} := D_{\phi,p,j} \cup \text{MAKEDEP}(\tau, \phi, eq_{\phi,p,j})$
ni
return $D_{\phi,p,j}$

Figure 3.1: Algorithms for deducing attribute dependencies from a conditional equation

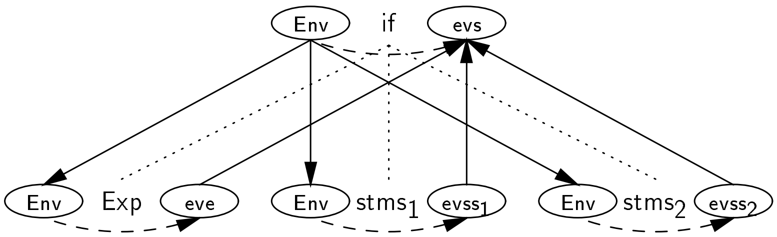


Figure 3.2: Attribute dependencies for an if-statement

3.3.2 Incremental reduction

When the user of the editor wants to know the ϕ-value of T, the term $\phi(T)$ is reduced incrementally. The incremental reduction algorithm we apply is based on a leftmost innermost strategy. Conditions are evaluated by reducing both sides and comparing the normal forms. A condition succeeds if these normal forms are equal.

Recall from the previous chapter the algorithm REDUCE-START (Figure 2.7) for incremental reduction in the non-conditional case. When a Φ-term or a parameter term is to be reduced we first check the associated attribute. If it contains a value and its status is not "unreliable", then we can skip reduction and replace the term by the stored attribute value. Otherwise, the term is reduced and the resulting normal form is stored in the attribute.

So after a term $\phi(T)$ has been reduced normal forms are stored in attributes of T. When subsequently the text in the editor is modified subtrees are replaced in T yielding T'. The algorithm PROPAGATE-TE-UP is applied to all synthesized attributes at the top of the new subtrees. It marks all their successors on dependency paths to the top of T' as "unreliable".

During rewriting of $\phi(T')$ unreliable attributes may get a new value. When an inherited attribute obtains a new value the algorithm PROPAGATE-TE-DOWN marks all successors at the top nodes of underlying trees as "unreliable".

3.3.3 Dynamically updating dependencies

In a non-conditional PRS all attributes in T that are predecessors of a synthesized attribute ϕ at the top of T obtain a correct value during reduction of $\phi(T)$. This is not necessarily the case in a conditional PRS. A synthesized attributes may have predecessors that do not affect its value. We optimize our algorithm by removing these dependencies dynamically.

Example 3.3 When reducing the term evs(if(a,assign(c,5),assign(c,7)),env(a:0,b:1,c:3)) the condition in equation [Evif1] succeeds. Hence the attributes in $Stms_2$ are not visited and do not obtain a value. As long as the value of the eve attribute remains 0, the attribute evss_2 is of no importance to evs. Nevertheless, a modification of $Stms_2$ will cause the evss_2 and its successor evs to be marked as "unreliable". Removing the edge between the evss_2 and evs will prevent needless re-evaluation. Figure 3.4 shows the resulting dependencies. □

The algorithm SELECT-EQ in Figure 3.3 is used by REDUCE for the selection of a matching equation. During this selection the dependency graph is updated dynamically.

Before a Φ-term is reduced, the incoming edges of the associated attribute are removed from the dependency graph D_T. When one of the sides of the condition is to be reduced, the dependencies that can be derived from this term are added to D_T. This effectively means that only incoming edges for the synthesized attribute at hand are being added. If all conditions succeed, the equation is to be applied for further

```
# In this algorithm equations are considered global information.
# REDUCE is the algorithm shown in Figure 2.7 on page 27.

SELECT-EQ(term,T)
let T = an attributed tree
    T_i  = the i-th subterm of T
    term = φ(S_i, y_1, ..., y_m) s.t. S_i equals T_i
    φ = the synthesized attribute at the top of T_i
    D_T = the dependency graph of T
    matching-equation = an  equation
in
    Remove incoming edges of φ
    matching-equation := none
    until a matching-equation is found
    do
        select an equation eq_{φ,p,j} ∈ EQ_{φ,p} :
        let eq_{φ,p,j} = (λ_1=ρ_1,...,λ_k=ρ_k) / (φ(p(x_1,...,x_n),y_1,...,y_m)=τ)
            condcheck, i = variables
        in
            condcheck := succeeded
            i := 1
            until condcheck = failed or i > k
            do
                let inst-λ_i, reduced-λ_i, inst-ρ_i, reduced-ρ_i = terms
                in
                    add dependencies for λ_i to D_T
                    inst-λ_i := instantiate λ_i
                    reduced-λ_i := REDUCE(inst-λ_i, T)
                    add dependencies for ρ_i to D_T
                    inst-ρ_i :=instantiate ρ_i
                    reduced-ρ_i := REDUCE(inst-ρ_i, T)
                    if     reduced-λ_i ≠ reduced-ρ_i
                    then  condcheck := failed
                    else    i := i + 1
                    fi
                ni
            od
            when i > k    # all conditions succeeded
            do
                    matching-equation := eq_{φ,p,j}
                    add dependencies for τ to D_T
            od
        ni
    od
    return matching-equation
ni
```

Figure 3.3: Algorithms for selecting an equation to reduce a Φ-term

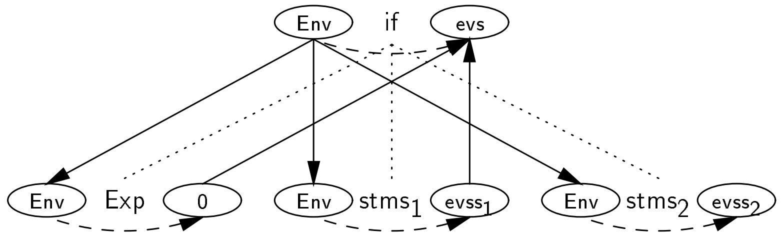


Figure 3.4: Attribute dependencies after evaluation of an if-statement, the value of evss_2 is irrelevant for evs.

reduction. Hence the dependencies derived from the right-hand side are added to D_T as well.

SELECT-EQ only updates the incoming edges of the synthesized attributes. Incoming edges of inherited attributes cannot be removed, nor do they cause useless computations. For instance, in Figure 3.4 the incoming edges for the inherited attributes Env at $Stms_2$ have not been removed. When the Env attribute of the if-node somehow gets a new value its successors at underlying nodes are marked as "unreliable", including Env and evss_2 at $Stms_2$. This is the only extra work caused by this dependency and it suffices. Leftmost innermost reduction induces a visiting order in which successor attributes of Env in stms_2 are visited only after Env has been computed. If its new value differs from the old value, successor attributes in $Stms_2$ are marked as "unreliable".

3.4 New variables in conditions

Thus far we assumed that the terms λ_i and ρ_i in conditions of a Φ-defining equation

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

are $S \cup \Phi \cup \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$-terms. We now look at the case where new variables can be introduced in conditions, i.e, variable that do not occur in the left-hand side of the conclusion of an equation.

Since we have implemented our incremental strategy as part of the term rewriting engine in the ASF+SDF-system, we look at the restrictions for new variables imposed by this system [Hen88].

New variables may be introduced in one side of a condition only. When the condition is evaluated all known variables are instantiated. Hence a closed term and an open term are obtained. The condition succeeds if the open term matches the normal form of the closed term. This yields instantiations of the new variables and the new variables are added to the list of known variables.

Open terms are not reduced. Therefore, the term containing new variables must be composed of free constructors, variables over sorts for which no equations exists, and new variables.

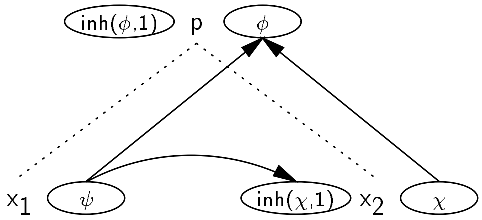


Figure 3.5: Attribute dependencies for equations with new variables

The algorithm MAKEDEP for deducing a dependency graph from an equation is adapted as follows. When a new variable is introduced no dependencies can be derived from it. When a variable that has been introduced in a condition is used in a later condition or in the right-hand side, we replace this occurrence by the other side of the condition in which it has been introduced. Then derivation of dependencies proceeds by the algorithm MAKEDEP as described before.

Example 3.4 In the following two equations *Newvar* is a variable different from x_1, x_2 or y. Both equations yield the dependency graph shown in Figure 3.5

$$\frac{\psi(x_1) = \mathit{Newvar}}{\phi(p(x_1, x_2), y) = \chi(x_2, \mathit{Newvar})} \qquad \frac{\psi(x_1) = \mathit{constr}(\mathit{Newvar})}{\phi(p(x_1, x_2), y) = \chi(x_2, \mathit{Newvar})}$$

□

It is clear that substituting *Newvar* is valid for new variables that are introduced as the complete term on one side of the condition. If *Newvar* is a subterm, like in the second equation, we should replace it in the right-hand side by something like $\chi(x_2, \mathit{proj}(\psi(x_1), 1))$, with $\mathit{proj}()$ a projection function. We can safely omit this projection because it is not a function in Φ and is therefore ignored in the construction of attribute dependencies.

3.5 Conclusions

We have described how incremental evaluation of terms can be extended efficiently to the larger class of conditional well-presented primitive recursive schemes. The method is based on dynamic updating of the dependencies, so that the only predecessors of an attribute at the top of a tree are those associated with equations that have been used to compute the normal form in that attribute.

Chapter 4

Incremental rewriting for list functions

We describe a method for incremental rewriting of functions over lists. The list functions belong to the class of so-called regular list functions. The method is a natural extension of the technique for incremental rewriting described in the previous chapters.

4.1 Introduction

The use of *list sorts* to specify linear lists improves the readability of a specification and has an intuitive appeal when specifying programming language constructs. We formulate properties of *regular* functions over lists so that these list functions can be viewed as a natural extension of primitive recursive schemes. Regular list functions give rise to regular attribute dependencies, so that the attribute dependency graph can be reconstructed efficiently after an insertion or deletion of elements in a list.

Overview of this chapter

In Section 4.2 we introduce and illustrate the usage of lists sorts in the ASF+SDF meta-environment. Section 4.3 introduces the basics of incremental rewriting for functions over lists sorts in such a way that it fits smoothly in our framework for incremental rewriting and that efficient insertion and deletion of list elements can be supported. In Section 4.4 we introduce the notion of *regular* list functions. Section 4.5 describes the incremental rewriting technique for regular list functions. In Section 4.6 we address the combination of list functions and conditional equations. Section 4.7 discusses related work. Conclusions are presented in Section 4.8.

4.2 Algebraic specifications with list sorts

The algebraic formalism ASF+SDF [BHK87, Kli93a] supports the use of list sorts [Hen89a], to define linear or associative lists. If the sort S has been declared in the signature of a specification, list sorts S* or S+, describing an iteration of respectively zero or more, or one or more elements of sort S, can be used as arguments in a function declaration. Variables over list sorts can be used in equations.

One advantage of list sorts is that it permits more natural specifications. Figure 4.1 and 4.3 both present a specification of a non-empty list of ELEMENTs with a concatenation function. The first one uses a binary operator to describe a list. The other one makes use of lists sorts and is shorter and more natural. In Figure 4.2 the abstract syntax trees for the concatenation of two lists are shown. One tree is for right-recursive lists, and the other one for linear lists.

Another advantage of the use of list sorts in a specification is that syntax-directed editors can give additional support for the editing of lists. Whereas the basic edit step on trees is replacement of a subtree by another subtree, edit actions on lists also include insertion or deletion of an element or a sublist.

4.3 PRSs with list sorts

We want to apply the technique for incremental rewriting as presented in the previous chapters, to primitive recursive schemes $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$ with functions Φ over list sorts in G.

Example 4.1 Figure 4.5 presents part of a specification of a type checker for a simple language. List sorts are used to specify declarations and statements. Figure 4.6 shows the attributes and attribute dependencies used for an incremental implementation of these type check functions. Attributes and dependencies in the top constructor of the tree are derived from equation [Tc1] in Figure 4.5. Other attribute dependencies will be discussed later. $\Box$

We give an intuitive description of incremental reduction of functions over lists. The resulting strategy should support the efficient behaviour of a syntax directed editor for G for insertion and deletion of list elements and sublists.

The basic model is that when a term $\phi(List, t_1, \ldots, t_m)$ is reduced, the normal forms of $t_1, \ldots, t_m$ are stored in inherited attributes of the abstract syntax tree of *List*. The normal form of the whole term is stored in a synthesized attribute at *List*. In the same way, normal forms for functions applied to a list element can be stored in attributes of that element.

Attributes are connected by a dependency graph, so that after a modification in the list affected attribute values can be marked as "unreliable". The dependencies between attributes must be "simple" in order to allow fast reconstruction of the list after insertion and deletion of elements. Therefore, we are interested in functions over lists that give rise to attributes and dependencies with the following properties:

- Only local attribute dependencies occur, that is, the value of an attribute of a list element depends only on attributes of neighbouring list elements and attributes of the whole list.

sorts: ELEMENT LIST
functions:
  single : ELEMENT $\to$ LIST
  list : ELEMENT $\times$ LIST $\to$ LIST
  conc : LIST $\times$ LIST $\to$ LIST
variables:
  *El*1, *El2* :$\to$ ELEMENT *List*1, *List2* :$\to$ LIST
equations:
[1]conc(single(*El*1), list(*El*2,*List*2)) = list(*El*1, list(*El*2,*List*2))
[2]conc(list(*El*1,*List*1), list(*El*2,*List2*)) = list(*El1*,conc(*List*1, list (*El*2,*List*2)))

Figure 4.1: Lists with concatenation, using right-recursive lists

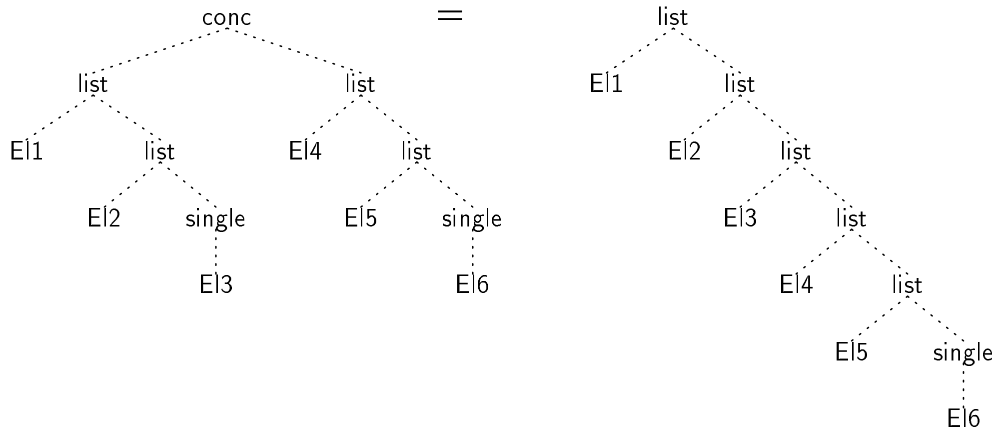


Figure 4.2: Abstract syntax trees for right-recursive lists.

sorts: ELEMENT LIST
functions:
  list_+ : ELEMENT+ $\to$ LIST
  conc : LIST $\times$ LIST $\to$ LIST
variables:
  *List*1,*List*2 :$\to$ ELEMENT+
equations:
[1]conc(list_+(*List*1), list_+(*List*2)) = list_+(*List*1,*List*2)

Figure 4.3: Lists with concatenation, using list sorts

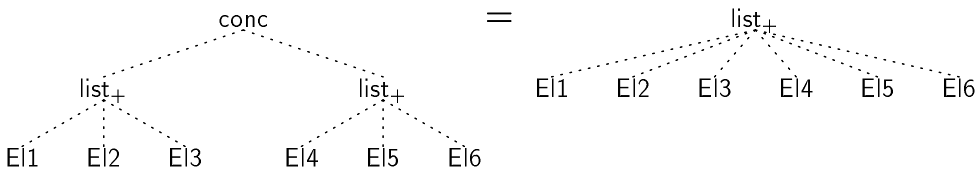


Figure 4.4: Abstract syntax trees for linear lists.

constructors:

prog	:	DECLS × STMS	→ PROGRAM
decls	:	DECL*	→ DECLS
stms	:	STM+	→ STMS

Φ-functions:

tcp	:	PROGRAM		→ BOOL
tcdecls	:	DECLS	× TENV	→ TENV
tcdecl	:	DECL	× TENV	→ TENV
tcstms	:	STMS	× TENV	→ BOOL
tcstm	:	STM	× TENV	→ BOOL

variables:

Decls	:→ DECLS	*Decl**	:→ DECL*	*Decl*	:→ DECL
Stms	:→ STMS	*Stm+*	:→ STM+	*Stm*	:→ STM
Tenv	:→ TENV				

Equations:

[Tc1] tcp(prog($Decls$,$Stms$)) = tcstms ($Stms$,tcdecls($Decls$,empty-env))
[Tc2] tcdecls(decls(), $Tenv$) = $Tenv$
[Tc3] tcdecls(decls($Decl$,$Decl$*), $Tenv$) = tcdecls(decls($Decl$*),tcdecl($Decl, Tenv$))
[Tc4] tcstms(stms(Stm,Stm+), $Tenv$) =
and(tcstm($Stm, Tenv$),tcstms(stms(Stm+), $Tenv$))
[Tc5] tcstms(stms(Stm), $Tenv$) = tcstm($Stm, Tenv$)

Figure 4.5: Part of a specification of a type checker. The specification has list sorts for declarations and statements.

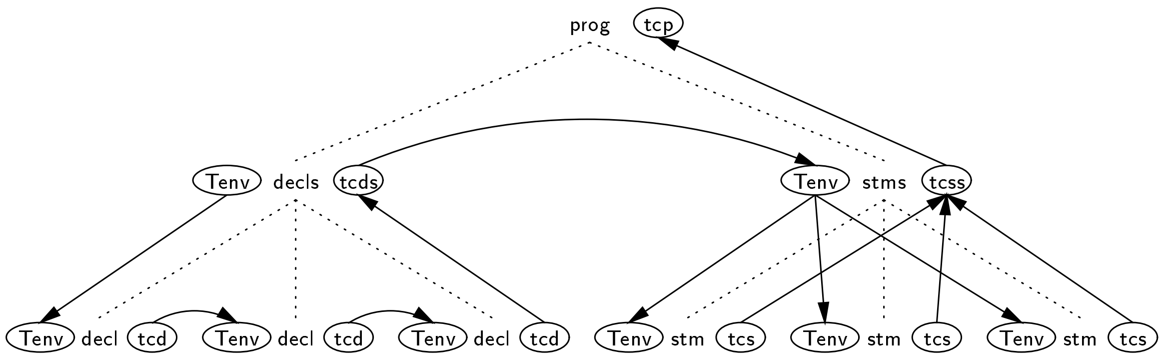


Figure 4.6: Part of an attributed tree with list nodes, some attribute names have been abbreviated: tcd=tcdecl, tcs=tcstm etc.

- The (top nodes of all) elements in the list except the first one and the last one have the same attribute dependencies,
- Dependencies of an element do not depend on the length of the list.
- The value of the inherited attributes of an element, or the value of the parameter terms of a function applied to an element, may not *directly* depend on the rank of that element, i.e, the ordinal position of the element.

Example 4.2 tcstms with equations [Tc4] and [Tc5] in Figure 4.5 is an example of a map-like function with desired dependencies. The dependencies that we will eventually derive for this function are depicted in Figure 4.6. When a statement is inserted, attribute dependencies for the new element can simply be added, the successor attribute tcstms of the new attribute tcstm is marked as "unreliable". □

Example 4.3 Equation [Tc3] in Figure 4.5, expresses that the value of the TENV parameter for a declaration depends on the result of applying the tcdecl-function to the previous declaration. One may argue that this reflects a dependency on the rank of a declaration. We do want to allow such dependencies because they can be expressed in the attribute dependencies, as is shown in Figure 4.6. When a declaration is inserted attribute dependencies between the neighbours must be removed and new attribute dependencies must be created. All direct and transitive successor attributes of the new attribute tcdecl are marked as "unreliable". □

Example 4.4 The list function χ and the element function ξ in the following equations exhibit an undesired behaviour.

$$\chi(list(El), y) = \xi(El, y) \tag{4.1}$$

$$\chi(list(El, List), y) = f(\xi(El, y), \chi(list(List), y + 1)) \tag{4.2}$$

When a term $\chi(list(El1, EL2, EL3), 10)$ is reduced according to this equation the result is $f(\xi(El1, 10), f(\xi(El2, 11), \xi(El3, 12)))$. The value of the parameter terms of the element function ξ is $\textit{Value-of-y} + (\textit{rank-of-element} - 1)$, hence it depends directly on the rank of that element. We want to exclude this kind of behaviour because when an element is inserted attributes of all elements to the right become unreliable. This dependency, however, is *not* expressed in the attribute dependencies, which means that the method for incremental rewriting for functions over trees as described in Chapter 2 can not be adapted directly to this situation. □

4.4 Regular functions over lists

In this section we formulate properties of equations defining functions over lists such that these functions "process elements in a regular way". We will call these functions *regular* list functions.

We assume that the list functions are members of Φ in a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$ with list constructors in G. Property (iii-list) given in Definition 4.1 below is the counterpart of (iii) formulated for equations over incremental functions over trees in Definition 2.3 in Chapter 2. Differences are that for a list function ϕ there are two defining equations in (iii-list) (rather than one equation), and that the decreasing property (iii-b), stating that the G-terms in the right-hand side are subterms of the G-term in the left hand side, is replaced by the list-decreasing property.

Due to (iii-list) the functions on elements in a list of length N $(N, M > 1)$ are invoked in the same way, namely via the right-hand side of the same equation as functions on elements in a list of length M $(N, M > 1)$.

Extra restrictions (list-a)–(list-c) are formulated in Definition 4.1 so that the functions over lists give rise to the desired regular attribute dependencies. Property (list-a) guarantees that a regular list function visits *all* elements of the list, and applies the same element function(s) to each element. Property (list-b) excludes list functions that mutually call each other, and thus may give rise to complicated attribute dependencies. Property (list-c) imposes restrictions on the parameters terms $v_1, \ldots, v_m$ in the term $\phi(list(Listx), v_1, \ldots, v_m)$ in τ_{list*} and τ_{list+} to ensure that the rank of an element does not *directly* influence the values of the parameters of the function applied to that element.

The requirements for the well-presentedness property are equal to those defined in Definition 2.4 in Chapter 2.

The equations have no conditions. In Section 4.6 we discuss the properties of conditional equations specifying regular list functions.

We use the constructor $list_*$ for a list with zero or more elements, $list_+$ for a list with one or more elements, and $list$ to indicate either $list_*$ or $list_+$ if it is clear from the context what kind of list is used.

Definition 4.1 (Regular list functions) Let $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$ be a well-presented primitive recursive scheme with parameters with lists sorts in G. A function ϕ over a list sort X_0 in G is *regular* if properties (iii-list), (list-a), (list-b) and (list-c) are met:

(iii-list) If ϕ is a function over the list sort X_0 and $list_*$ is a list constructor $list_* : X* \rightarrow X_0$ in G, then *two defining* equations $eq_{\phi,list}$ exists. One handling the empty list, the other one for lists of at least one element. In the latter equation, the list is represented by one variable over the list element X and one list variable.

$$\phi(list_*(), y_1, \ldots, y_m) = \tau_\emptyset \tag{4.3}$$

$$\phi(list_*(Elx, Listx), y_1, \ldots, y_m) = \tau_{list*} \tag{4.4}$$

With *Elx* an X variable and *Listx* an X* variable.

If $list_+$ is a list constructor $list_+ : X+ \rightarrow X_0$ in G the definition is similar: There is one equation defining ϕ over one element, and one for lists of at least two elements.

$$\phi(list_+(Elx), y_1, \ldots, y_m) = \tau_1 \tag{4.5}$$
$$\phi(list_+(Elx, Listx), y_1, \ldots, y_m) = \tau_{list+} \tag{4.6}$$

With *Elx* an X variable and *Listx* an X+ variable.

(a) Equations (4.3), (4.4), (4.5) and (4.6) are left linear.

(b) The first argument of a Φ-subterm in a right-hand side is *Elx* or $list*(Listx)$ (4.4) or $list+(Listx)$ (4.6). We say that they are *list-decreasing* in their first argument.

(list-a) In τ_{list_*} and τ_{list_+} the term $\phi(list(Listx), v_1, \ldots, v_m)$ occurs at least once.

When both $\phi(list(Listx), v_1, \ldots, v_m)$ and $\phi(list(Listx), v'_1, \ldots, v'_m)$ occur in τ_{list_*} or τ_{list_+} then $v_i = v'_i, 1 \leq i \leq m$.

(list-b) There are no occurrences of Φ-terms over *list(Listx)* other than $\phi(list(Listx), v_1, \ldots, v_m)$ in τ_{list_*} and τ_{list_+}.

(list-c) A parameter term $v_j, 1 \leq i \leq m$, in the right-hand side of τ_{list*} or τ_{list+} is either

- y_j, i.e, equal to the parameter in the same position on the left hand side. (See [Tc4] in Figure 4.5)
- or a term composed of function symbols, constants, and Φ-terms over the element *Elx*, $\phi(Elx, w_1, \ldots, w_k)$ terms. (See [Tc3] in Figure 4.5)

□

4.5 Incremental implementation of list functions

For incremental rewriting of terms with list terms, we store information in attributes at nodes in the list. With each function ϕ that applies to the list term, a synthesized attribute is associated as well as an inherited attribute for each parameter of ϕ. In the same way attributes are associated with the functions on elements of the list. As before attributes have a value and a status. The status indicates if the value can safely be used or not.

Assume that *Listx* is an abstract syntax tree of a list with attributes for the list function ϕ and for functions over the elements. When the term $\phi(Listx, t_1, \ldots, t_m)$ is to be reduced, we first inspect the attribute for ϕ. If the status indicates that the attribute value is reliable, the value can be used to avoid reduction. Otherwise, the term is reduced and its normal form is stored in the attribute. The status is then reset to "reliable". The same procedure is applied for incremental reduction of the parameter terms t_i, and for incremental reduction of the Φ-terms and parameter terms for element functions $\psi(El, s_1, \ldots, s_k)$.

4.5.1 Attribute dependencies

Attributes are connected by means of attribute dependencies, so that status information can be propagated. We derive attribute dependencies for a regular list function ϕ from its defining equations. When deriving attribute dependencies we make use of *imaginary* attributes for sublists. An imaginary attribute is never a predecessor of another attribute *Att*. Instead, the predecessors of the imaginary attribute become the predecessors of this attribute *Att*.

The incoming edges for the synthesized attribute ϕ of a list, i.e, attribute ϕ of the complete list or the *imaginary* attribute ϕ of its sublists, are derived from the right-hand sides τ of its defining equations.

- Let y_i be a parameter of the left-hand side of the defining equations. If y_i is a subterm of τ and y_i is not a subterm of a Φ-term, then ϕ directly depends directly on the inherited attribute, $\mathsf{inh}(\phi, i)$ for the i-th parameter of ϕ.

- If $\psi(El, w_1, \ldots, w_k)$ is a subterm of the right-hand side and it is not a subterm of another Φ-term, then ϕ directly depends directly on the synthesized attribute ψ of the first element.

- If $\phi(\mathsf{list}(List), v_1, \ldots, v_m)$ is a subterm of the right-hand side and it is not a subterm of another Φ-term, then the predecessors of ϕ at the top of the list are the predecessors of the (imaginary) ϕ attribute for the sublist.

Let $\psi(El, w_1, \ldots, w_k)$ be a subterm of the right-hand side of a defining equation. The incoming edges for the inherited attribute $\mathsf{inh}(\psi, j)$ associated with the j-th parameter of the function ψ over list elements, are derived from the subterm w_j.

- Let y_i be a parameter of the left-hand side of the defining equations. If y_i is a subterm of w_j and y_i is not a subterm of a Φ-term, then $\mathsf{inh}(\psi, j)$ of the first list element depends on the inherited attribute, $\mathsf{inh}(\phi, i)$ for the i-th parameter of ϕ.

  $\mathsf{inh}(\psi, j)$ of any other elements depends on the predecessors of imaginary inherited attribute, $\mathsf{inh}(\phi, i)$ of the sublist. Because of property (list-c) in Definition 4.1 $\mathsf{inh}(\phi, i)$ of the sublist either has one predecessor, namely $\mathsf{inh}(\phi, i)$ of the complete list, or its predecessor is the ψ attribute of the left neighbour.

- If $\phi(\mathsf{list}(List), v_1, \ldots, v_m)$ is a subterm of w_j and it is not a subterm of another Φ-term, then $\mathsf{inh}(\psi, j)$ depends on the predecessors of the imaginary synthesized attribute ϕ of the sublist.

Example 4.5 From equation [Tc5] in Figure 4.5 we derive that the tcstms of the list of statements depends on the tcstm attribute of each element. The inherited attribute TENV for tcstm of each element depends on the inherited attribute TENV of the list of statements, because the imaginary inherited attribute TENV of each sublist depends on the inherited attribute TENV of the complete list. □

Example 4.6 From equations [Tc2] and [Tc3] in Figure 4.5 we derive that the tcdecls of the list of declarations depends on the tcdecl attribute of the last element. The TENV attribute of the first element depends on the TENV attribute of the whole list. The TENV attribute of any other element depends on the tcdecl attribute of its left neighbour, because the imaginary synthesized attribute tcdecls of each sublist depends on the synthesized attribute tcdecl of its left neighbour. □

4.5.2 Attribute updating

When a list element is modified, its top attributes are marked as "unreliable" and so are its direct and transitive successor attributes.

When an element has been inserted in or deleted from a list, all relevant attribute dependencies of the new element and/or its neighbouring elements must be restored. If an attribute is new or a new incoming edge is added to an attribute, or an incoming edge is removed from an attribute, this attribute is marked as "unreliable", and so again this is propagated to all its successors.

When after one or more modifications in *Listx*, $\phi(Listx', t_1, \ldots, t_m)$ is to be reduced, the values of reliable attributes can be used to avoid reduction steps.

4.6 Conditional equations

In Chapter 3 we introduced conditional incremental rewriting for functions over trees. For conditional equations for list functions Definition 4.1 is adapted as follows. In property (iii-list) more than two equations can be given for a list function ϕ. Properties (list-a)–(list-c) must be valid for the union $\bigcup_{Eq_\phi}(\bigcup_i \lambda_i \cup \rho_i) \cup \tau_{list}$, with λ_i and ρ_i conditional terms and τ_{list} the right-hand sides of equations with left hand side $\phi(list(Elx, Listx), y_1, \ldots, y_m)$.

4.7 Related Work

In [Jeu91] and [YS91] incremental implementations for list functions are described based on finite differencing. In both papers the point of departure is different from ours. We describe the incremental implementation for a user-defined list function whereas they assume that for any list function additional functions are given that help incremental evaluation.

Jeuring [Jeu91] proposes a method for description and derivation of algorithms in interactive systems (editors) for incremental computation of functions over lists. In his edit model the result of the function application *f(list)* of a list in the editor is recomputed after each edit action. To this purpose, the results *f(left-sublist)* and *f(right-sublist)* for the sublists on both sides of the cursor, are updated after each cursor replacement and after each edit action. He assumes the existence of an operator $\odot$ such that for the concatenation # of two lists holds $f(\textit{left-list} \mathbin{\#} \textit{right-list}) = f(\textit{left-list}) \odot$

f(*right-list*). The function $\odot$ is used to incrementally compute the new value of *f(left-sublist)* after an element has been added or deleted, and to compute *f(list)* from the sublist results.

INC [YS91] is a language for incremental computation designed by Yellin and Strom and based on finite differencing. In an INC program a function over lists is the composition of predefined functions. An incremental behaviour for functions over lists or other data-types is automatically available since for each predefined function f, functions *f-add* and *f-delete* exist. After an element has been added or deleted from a list these functions are used to incrementally compute the new f value.

The Synthesizer Generator is a programming environment generator based on attribute grammars [RT89a, RT89b]. In its specification language SSL, phyla (sorts) can be declared as list phyla. The effect of this declaration is an adaptation in the editor for insertion and deletion of list elements, so that it looks as if list phyla are linear lists. Restrictions in SSL for a list phylum are that two production rules must be given, one nullary production to specify the empty list and one binary production specifying a right-recursive list. This coincides with (iii-list) in Definition 4.1. The main difference with our approach is that lists in the Synthesizer Generator are internally represented as right-recursive lists, which means that the standard rules for attribute dependencies and attribute updating in trees need not be adapted for lists, hence no special requirements for attribute definition rules over lists have to be formulated, like (list-a)–(list-c) in Definition 4.1.

4.8 Conclusions

We have described a technique for incremental evaluation of functions over list sorts. The method is a straightforward adaptation of the technique for incremental reduction of functions over trees, as defined in the previous chapters. Restriction on the defining equations for list functions are formulated so that functions over lists sorts are a natural extension to PRSs, and that the derivation of attribute dependencies in lists is simple and yields regular dependencies in a list.

In an ASF+SDF specification one can of course choose between specifying lists as linear lists by means of a list sort, or as right-(or left-) recursive lists by means of a binary constructor.

In general, we find that with respect to incremental implementation, linear lists are to be preferred over right-recursive list. Occasionally, however, right-recursive lists lead to more efficient incremental performance.

Linear lists offer efficient incremental computation after insertion or deletion of elements, because the abstract syntax tree for the tail of the list is preserved and so is all attribute information in elements in the tail of the list. For right-recursive lists more information is stored: namely in attributes of the sublists. When attributes values of an element do not affect attribute values in elements to the right (in the tail of the list), right-recursive lists may offer more efficient incremental computation after an element is modified, because the attribute values at the tail of the list can be re-used. The effect depends on the position of the element and the costs of the computation of sublist results.

The class of regular functions is smaller than the class of functions for right-recursive lists with an incremental operator. We are sure, however, that these limitation cause very little problems in practice.

Chapter 5

Fine-grain incremental rewriting

In the previous chapters we have described how an incremental implementation can be derived from algebraic specifications belonging to the subclass of well-presented primitive recursive schemes. We combined term rewriting with techniques for storage and re-use borrowed from attribute grammars. The uniformity of algebraic specifications allows us to generalize our incremental algorithms to functions on values of auxiliary data types, without extending or modifying the specification language. Thus, we can 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*.

A fine-grain incremental implementation of a table data type can, for instance, solve the problem caused by aggregated attribute values like symbol tables. (A change in an aggregated value causes a re-evaluation of all attributes that depend on this table, even if they do not depend on the modified part.) When a lookup function in a table data type is declared to be incremental, a function cache for the lookup function is automatically generated. Direct *definition-use* dependencies are established from entries in the function cache to attributes in the edit-tree that functionally depend on these entries. Moreover, after each modification in the table the cache is updated incrementally.

5.1 Introduction

In Chapter 2 we have 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 algorithm is based on the equivalence between such PRSs and strongly non-circular attribute grammars [CFZ82]. For each *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. While the term typecheck(Program) is being reduced, normal forms of intermediate type check terms are stored in attributes attached to subterms of Program. Editing the program corresponds to one or more subterm replacements and is followed by an updating of the status of attributes to indicate which attributes have become unreliable. In a subsequent reduction the values of reliable attributes can be used to avoid reduction steps, whereas other attributes obtain a new type check value.

A first implementation of this method in a programming environment generator based on algebraic specifications shows that incremental evaluation is often advantageous, but that attributes containing aggregated values, like symbol tables, give rise to very inefficient incremental evaluation.

Typically, a symbol table is used to gather information from places in a program where variables are defined, and this information is propagated through the program to all attributes at places where variables are used. A modification of a part of the symbol table causes all these use-attributes to be re-evaluated, rather than just the ones depending on the modified component. In this chapter we present a solution based on a fine-grain incremental implementation of algebraic specifications.

Algebraic specifications offer a uniform way of describing data types, and the abstract syntax and semantics of a language are just examples of such data types. So, in principle an incremental implementation can be generated for each data type as long as its specification is a PRS. We use this idea to refine our incremental algorithm. The main thing we need to do is find out how attributed terms of auxiliary data types can be connected to the attribute graph of a tree of the main PRS, and how their incremental implementation can be used efficiently.

The fine-grain incremental implementation method can typically be applied to a type check specification with a table data type with an *incremental* lookup function. In such a case, a function cache for lookups in this particular table is generated. Direct *definition-use* dependencies are established from entries in this function cache to functionally dependent attributes in a program tree. The cache is updated incrementally when a modification in the symbol table has been detected during attribute evaluation.

The approach we present is general in the sense that it can be applied to any specification in the class of *layered primitive recursive schemes*, without extending the specification formalism or adding predefined data types. This distinguishes it from techniques in the field of attribute grammars. In attempts to solve the problem of aggregate values in attribute grammars, either a predefined data type is added to the specification formalism or the attribute grammar formalism is extended or combined with another formalism.

Overview of this chapter

Section 5.2 briefly explains and illustrates primitive recursive schemes $\langle G, S, \Phi, Eq, Eq_\Phi\rangle$ with non-disjoint G and S. Our technique for fine grain incremental evaluation makes use of *copy dependencies*. Section 5.3 repeats the basic steps of incremental evaluation, and explains incremental evaluation in the presence of copy dependencies. Sections 5.4 to 5.6 constitute the heart of this chapter. In Section 5.4 we show how incremental reduction can be applied to the lookup function of a table data type.

Section 5.5 presents the definition of layered primitive recursive schemes and describes how fine-grain incremental evaluators are derived from specifications belonging to this class. In Section 5.6 we describe the resulting incremental rewriting strategy. Both the basic incremental technique and the fine-grain extension have been implemented as part of the term rewriting engine of a programming environment generator. In Section 5.8 we discuss related work. Finally, Section 5.9 contains some conclusions.

5.2 Primitive recursive schemes with non-disjoint G and S

We briefly recall the notion of primitive recursive schemes as introduced in Chapter 2.

A primitive recursive scheme with parameters (PRS) 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. $Eq \cup Eq_\Phi$ are the equations of the specification. 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, as well as the output sort are sorts of the signature S of the auxiliary data types.

Example 5.1 Figure 5.1 presents part of an algebraic specification of the type checker of a simple programming language. This specification is a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$. G consists of the sorts PROGRAM, DECLS, DECL, STMS, STM and the functions program, decls, stms, stm, assign. Not shown here are the functions of sorts BOOL,TENV and TYPE that form S, nor the associated equations, which are elements of Eq. The type check functions, tcp, tcdecls, tcdecl, tcstms and tcstm are the Φ-functions or *incremental* functions. [Tc1]–[Tc6] are the Φ-defining equations.

When declarations are type checked a type-environment is constructed. This is a symbol table with identifiers and their associated types. A type-environment is used as a second argument for type checking both declarations and statements. Hence, TENV is a *parameter* of these functions. The signature of the type-environments will be presented in Section 5.4. □

In Definition 2.3 of a PRS the sorts of S and G are assumed to be disjoint. Yet, in existing type check 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 equation [Tc6] in Figure 5.1 the variable *Id* appears in the G-term assign(*Id*,*Exp*), but also in the S-term compatible(*Tenv*,*Id*) in the right-hand side.

In this chapter, we allow an overlap of G and S under the restriction that no equations exist for the shared sorts. Chapter 6 discusses technical aspects of non-disjoint G and S in a PRS.

```
sorts: PROGRAM DECLS DECL STMS STM EXP BOOL TENV

constructors:
        program     : DECLS × STMS  → PROGRAM
        empty-decls :               → DECLS
        decls       : DECL  × DECLS → DECLS
        stms        : STM   × STMS  → STMS
        stm         : STM           → STMS
        assign      : ID    × EXP   → STM
Φ-functions or incremental functions:
        tcp     : PROGRAM        → BOOL
        tcdecls : DECLS   × TENV → TENV
        tcdecl  : DECL    × TENV → TENV
        tcstms  : STMS    × TENV → BOOL
        tcstm   : STM     × TENV → BOOL

variables:
        Decls :→ DECLS    Decl :→ DECL    Stms :→ STMS
        Stm   :→ STM      Exp  :→ EXP     Id   :→ ID
        Tenv  :→ TENV

Φ-defining equations:
[Tc1]  tcp(program(Decls,Stms)) = tcstms(Stms,tcdecls(Decls,empty-env))
[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]  tcstms(stm(Stm), Tenv) = tcstm(Stm, Tenv)
[Tc6]  tcstm(assign(Id,Exp), Tenv) = compatible(lookup(Tenv,Id),tcexp(Exp, Tenv))
```

Figure 5.1: Part of an algebraic specification of a type checker

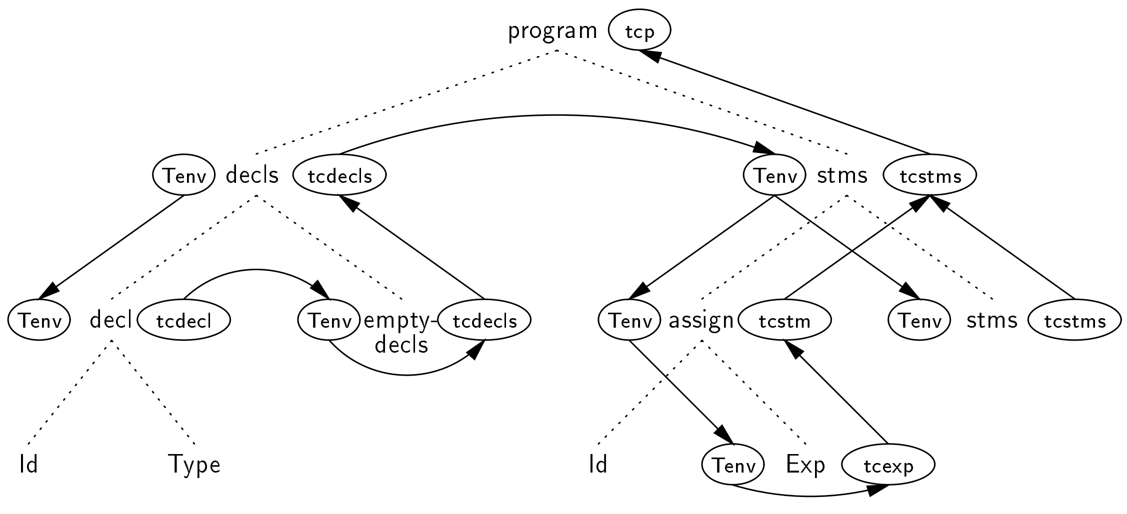


Figure 5.2: Top of an attributed term

5.3 Incremental rewriting with copy dependencies

For fine-grain incremental evaluation we will make use of copy dependencies between attributes in a decorated tree. First, we briefly recall our standard technique for incremental rewriting as described in Section 2.5.2. Then we introduce copy attributes and copy dependencies. Finally, we discuss the adaptation needed for incremental rewriting in the presence of copy dependencies.

5.3.1 Incremental rewriting

Assume that $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$ is a PRS. Assume furthermore that T is a G-term and the attributed abstract syntax tree of a text in an editor. When the user of the editor wants to know the ϕ value of T, the term $\phi(T)$ is reduced. All attributes that are visited during reduction of $\phi(T)$ obtain a value.

During subsequent editing subtrees are replaced in T. After each subtree replacement the synthesized attributes at the replacement node are marked as "Initial" and for each synthesized attribute the algorithm PROPAGATE-TE-UP of Figure 2.11 is invoked to reset the status of its successors along a *upward* and *horizontal* dependency path to the root of T to "TobeEvaluated".

When the user of the editor wants to know the new ϕ value after several edit steps (i.e. after several subtree replacements) the term $\phi(T')$ is reduced incrementally. Rewriting of a Φ-term or a parameter term is preceded by a check on the associated attribute. If the status of the attribute indicates that the attribute contains a reliable value, reduction can be avoided and the attribute value can serve as the normal form of the term to be reduced. If the attribute does not contain a reliable value, the term is reduced and its normal form is stored in the attribute. The new attribute value is then compared with the previous one. If both values are the same the status of the attribute is set to "Unchanged", and its successors are inspected by the algorithm PROPAGATE-UC of Figure 2.13: If a successor has status "TobeEvaluated" and it has only "Unchanged" predecessors its status becomes "Unchanged" as well.

If the new value differs from the previous one, the status of the attribute is set to "Changed", and PROPAGATE-TE-DOWN of Figure 2.12 is invoked to mark all downward successors in the underlying production as "TobeEvaluated".

5.3.2 Copy dependencies

Often the value of an attribute B is the copy of the value of its predecessor A. In that case we will call B a *copy attribute*. If during incremental reduction an attribute gets a new value the new value is compared to the previous one. This may be an expensive operation. The efficiency of incremental evaluation is improved when copy attributes share a value with their predecessors, because the old value and the new value has to be compared only once for all copies.

Deriving copy dependencies

Attribute dependencies are derived from Φ-defining equations. It is simple to derive copy dependencies as well. Consider the constructor $p : X_1 \times \ldots \times X_n \rightarrow X_0$ and the equation $\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$, with $\psi(x_i, v_1, \ldots, v_k)$ a subterm of the right-hand side τ.

If the head symbol of τ is a Φ-function, i.e, τ equals $\psi(x_i, \ldots)$, then the value of the synthesized attribute for ϕ is a copy of the value of the synthesized attribute for ψ of x_i.

Let $\mathsf{inh}(\psi,\mathsf{l})$ $(1 \leq l \leq k)$ be the inherited attribute of X_i associated with the l-th parameter of ψ. Copy predecessors are derived for $\mathsf{inh}(\psi,\mathsf{l})$ by analysing v_l as follows. If $v_l = y_j$ $(1 \leq j \leq m)$ $\mathsf{inh}(\psi,\mathsf{l})$ is a copy of $\mathsf{inh}(\phi,\mathsf{j})$ of X_0 the inherited attribute associated with the j-th parameter of ϕ. If $v_l = \xi(x_j, \ldots)$ $(1 \leq j \leq n)$ $\mathsf{inh}(\psi,\mathsf{l})$ is a copy of the synthesized attribute ξ of X_j.

Example 5.2 From equations [Tc1] in the type check example of Figure 5.1 we derive that the tcp attribute of a program is a copy of the tcstms attribute of its statements, and that the type-environment attribute of the statements subtree of a program, is a copy of the tcdecls attribute of the declarations. From equations [Tc4]–[Tc6] it can be concluded that all type-environment attributes of a tree in the statements section are copies of the type-environment attributes at the ancestor of that tree. □

5.3.3 Rewriting with copy dependencies

When a tree is decorated with attributes chains of copy attributes can appear. The *head* of a copy chain in an attribute graph is a non-copy attribute with copy successors. We refer to the direct non-copy successors of the copy attributes as *use* attributes.

If an attribute in a tree is a copy of its predecessor they share the same value and the same status. This means that when a *copy-head* attribute obtains a new value, copies automatically obtain the same new value and the same new status. The non-copy successors of these copies however have to be marked as "TobeEvaluated" as well. This requires dependencies between a *copy-head* and its *use* attributes, as well as an additional algorithm for remote status propagation.

We use an approach inspired by Hoover [Hoo86] for creating remote dependencies between a *copy-head* and its *use* attributes. When a tree is decorated each copy attribute obtains information about the *copy-head* and the path from the *copy-head* to itself. The *copy-head* keeps a so called *copy bypass tree* of all copies that have non-copy successors They are ordered according to their paths.

A direct, *non-local*, dependency is established from the copy-head to its *use* attributes.

Example 5.3 In Figure 5.2 the lower tcdecls attribute is a *copy-head*. Its *use* attributes are the synthesized attributes tcstm at the assign nodes, because they are non-copy successors of the inh(tcstm,1) attributes.

```
PROPAGATE-TE-REMOTE(copy-head)
let copy-head = a copy-head  attribute with status "Changed" in a dependency graph D_T
      use, ψ = attributes
in
  for all use attributes of copy-head      # non-copy successors of a copy of copy-head
    do when  status_use ≠ "TobeEvaluated" or "Initial"
           do  status_use := "TobeEvaluated"
                if  use is a synthesized attribute
                then
                      PROPAGATE-TE-UP(use)
                else
                      use is an inherited attribute
                      for  all its corresponding synthesized attributes ψ
                       do  status_ψ := "TobeEvaluated"
                            PROPAGATE-TE-UP(ψ)
                       od
                fi
           od
    od
ni
```

Figure 5.3: Algorithm for propagation "TobeEvaluated" for the non-copy successors of the copies of a Changed *copy-head*.

Let (1,2) denote a path indicating: down to the first subtree, then down to the second subtree. Let $\mathsf{inh}(\mathsf{tcstm}, 1)_{\mathsf{assign}}$ denote the inherited attribute for the first parameter of the function tcstm at the node assign, etc. The lower tcdecls attribute in Figure 5.2, maintains the following *copy bypass tree*

```
(1)
(2) (2,1) (2,1,1)-inh(tcstm,1)_assign
          (2,1,2)
```

□

The algorithm PROPAGATE-TE-REMOTE presented in Figure 5.3 marks each *use* attributes of a *copy-head* as "TobeEvaluated" and invokes PROPAGATE-TE-UP for these *use* attributes.

5.3.4 Keeping non-local dependencies consistent

During editing subtrees are replaced in the abstract syntax tree T, and attributes as well as their incoming and outgoing edges are removed from the attribute dependency graph D_T. After a replacement of a subtree *Oldsub* with a *copy-head* attribute or with *use* attributes, some care must be taken in removing the old non-local dependencies.

When a subtree has been removed and one of nodes contains a *copy-head* attribute there are two possibilities: If all its *use* attributes are inside *Oldsub* these *use* attributes are automatically removed together with the non-local dependencies. If, on the other hand, a *use* attribute exists outside *Oldsub*, a copy attribute at the top node of the *Oldsub* exists that passes on information from the *copy-head* to this *use* attribute. Let us call this attribute *replacement-copy*. The replacement-copy contains a reference to the *copy-head* with the *copy bypass tree*. First, the *copy bypass tree* is pruned at the copy path of the replacement-copy. We then use the copy attributes in this pruned subtree to find the *use* attributes of *copy-head* outside *Oldsub* and remove their remote incoming edges.

When a subtree *Oldsub* has been removed and some node inside this subtree contains a *use* attribute, there are again two possibilities: If the related copy-head is inside the old subtree as well it is removed together with its non-local dependencies. If, on the other hand, the related copy-head is not inside *Oldsub* again a *replacement-copy* exists at at the top node of *Oldsub*. The replacement-copy contains a pointer to the *copy-head*. First, the *copy bypass tree* is pruned at the copy path of the *replacement-copy*. We then use the copy attributes in this pruned subtree to find the *use* attributes in *Oldsub* and remove their remote incoming edges.

New non-local dependencies can be created when *Newsub* is decorated with attributes.

5.4 Table + lookup as PRS

We now return to the main theme of this chapter. For most specifications of type checkers incremental reduction with or without copy-dependencies is inefficient after a modification in the declaration section. Figure 5.2 shows that when the declarations are modified, a component in the table in the top tcdecls-attribute is likely to have changed. All attributes in the statement section are successors of this aggregate attribute, hence they all become unreliable and no type check results of the statements can be reused.

We will solve the inefficiency caused by multiple dependencies of an aggregate value by extending the incremental technique to functions like the lookup function of the type-environment.

Example 5.4 In Figure 5.4 part of the specification of a type-environment is presented. This algebraic specification is a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$. G is formed by the sorts TENV, PAIR, TYPE and ID, and the constructors pair, empty-env and tenv. The constructors for TYPE have been omitted. lookup, id-of and type-of are the elements of Φ. Equations [Tenv1]–[Tenv5] are the Φ-defining equations ID and TYPE also serve as the auxiliary data types S, that are used as the output sorts and the parameters of the Φ-functions. Eq is empty. □

For incremental evaluation of lookup(Tenv,Id) we can store the type-environment and decorate it with Id and lookup attributes, as shown in Figure 5.5. After changing Tenv to Tenv', affected attributes are marked as unreliable. When reducing lookup(Tenv',Id), the normal forms in reliable attributes can be reused.

5.5 Layered PRSs and auxiliary attributed terms

In this section we define the class of layered primitive recursive schemes. We explain in two steps how attributed terms of auxiliary data types like the type-environment can be connected to the attribute graph of the abstract syntax tree of the text in an editor, like Program. First, we introduce so-called *auxiliary attributed terms*, and explain how they are used for a fine-grain incremental implementation of layered PRSs. We will show that this technique efficient mainly when used in combination with copy attributes in the abstract syntax tree of the text in an editor. This leads us to the definition of *multiply attributed* (auxiliary) terms with function cache.

5.5.1 Layered primitive recursive schemes

Informally speaking, a layered PRS is a PRS which contains another PRS. This sub-PRS in turn can be layered as well. More formally:

Definition 5.1 (Layered PRS) 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

- The specification of the auxiliary data types $\langle S_1, Eq_1\rangle$ is also a (possibly layered) PRS $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$.
- If a Φ_2-subterm occurs in the right-hand side τ of a Φ_1-defining equation
$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$$
then the first argument of this Φ_2-subterm is *not* x_i $(1 \leq i \leq n)$.

□

Note that in a layered PRS $S_1 = G_2 \cup S_2 \cup \Phi_2$ and $Eq_1 = Eq_2 \cup Eq_{\Phi_2}$.

The second requirement is needed because we allow a limited overlap between S and G (Section 5.2). A consequence of this requirement is that, when during reduction of a Φ_1-term with a first argument T, a function $\phi_2(U, \ldots)$ of a different layer is introduced, recursion will shift from T to the new structure U.

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.

Example 5.5 The specification formed by the union of the type checker in Figure 5.1 and the type-environment in Figure 5.4 is a well-presented layered PRS, with G_1 formed by the sorts PROGRAM,DECLS,DECL,STMS,STM etc. and the related constructor functions. Φ_1 are the functions tcp, tcdecls, tcdecl, tcstms and tcstm. G_2 is formed by the sorts TENV, PAIR, TYPE, ID and the constructors pair, tenv and empty-env and the Φ_2-functions are lookup, id-of, and type-of. □

sorts: TENV PAIR TYPE ID

constructors:

pair	:	ID	$\times$ TYPE	$\to$ PAIR
empty-env	:			$\to$ TENV
tenv	:	PAIR	$\times$ TENV	$\to$ TENV

Φ-functions or incremental functions

lookup	:	TENV	$\times$ ID	$\to$ TYPE
id-of	:	PAIR		$\to$ ID
type-of	:	PAIR		$\to$ TYPE

Φ-defining equations:

[Tenv1] $\dfrac{\text{id-of}(\mathit{Pair}) = \mathit{Id}}{\text{lookup}(\text{tenv}(\mathit{Pair}, \mathit{Tenv}), \mathit{Id}) = \text{type-of}(\mathit{Pair})}$

[Tenv2] $\dfrac{\text{id-of}(\mathit{Pair}) \neq \mathit{Id}}{\text{lookup}(\text{tenv}(\mathit{Pair}, \mathit{Tenv}), \mathit{Id}) = \text{lookup}(\mathit{Tenv}, \mathit{Id})}$

[Tenv3] lookup(empty-env, Id) = error-type
[Tenv4] id-of(pair($\mathit{Id}, \mathit{Type}$)) = Id
[Tenv5] type-of(pair($\mathit{Id}, \mathit{Type}$)) = Type

Figure 5.4: Part of an algebraic specification of a type-environment

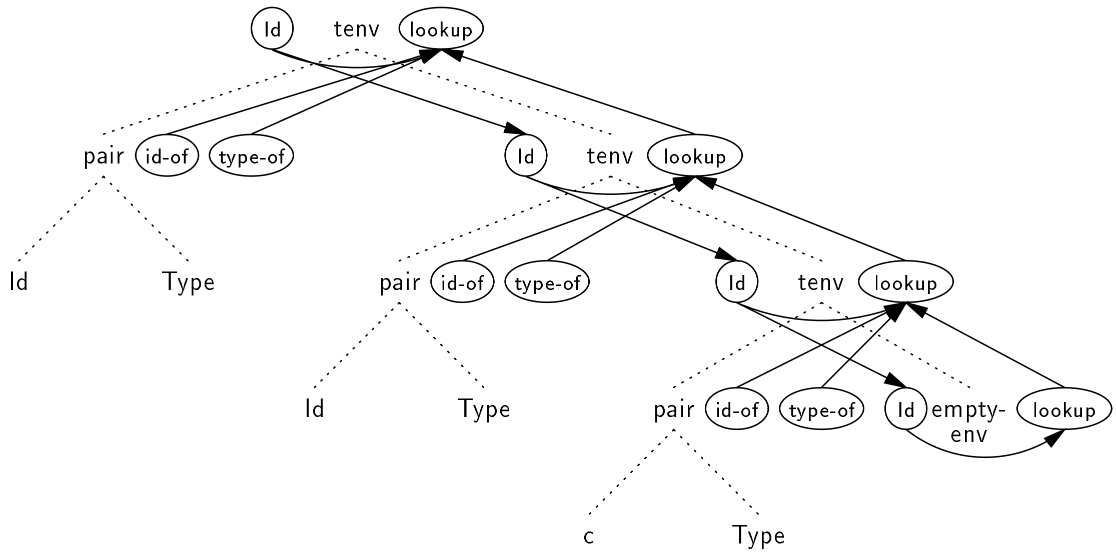
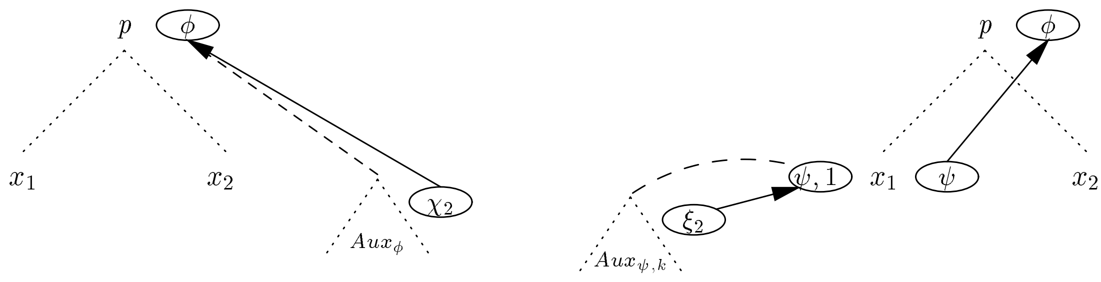


Figure 5.5: An attributed type-environment



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

Figure 5.6: Auxiliary attributed terms

5.5.2 Auxiliary attributed terms

We will use the properties of a layered PRS to get incremental evaluation of the value within Φ_1-attributes, by storing Φ_2-terms that occur in the reduction of a Φ_1-term as *auxiliary attributed terms*: G_2-terms decorated with Φ_2-attributes.

Auxiliary attributed terms can be related to both inherited and synthesized attributes.

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 its right-hand side τ.

- If the Φ_2-term is *not* a subterm of a Φ_1-term, an auxiliary term is related to the synthesized attribute of function ϕ (Figure 5.6a).
- 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. An auxiliary term is then associated with the inherited attribute of the k-th parameter of ψ (Figure 5.6b).

The basic idea of incremental reduction with auxiliary attribute terms is now as follows. Auxiliary terms obtain their initial value and are attributed upon reduction of the related Φ_1-term. If the auxiliary term already exists, the differences between the old value and the new value must be computed and subtrees in the old value are replaced to obtain the new value. Next, affected attributes of the auxiliary term are marked as unreliable and so are their direct and transitive successors in the edit tree.

Hence, reduction of a Φ_2-term is replaced by a tree difference calculation followed by an updating of the status of attributes.

Clearly, incremental reduction with auxiliary attribute terms can only be profitable if the tree difference calculation is very cheap or if many Φ_1-attributes share one auxiliary term, so that this calculation has to be performed only once. We, therefore, focus on layered PRSs in which auxiliary terms are the values of *copy attributes*.

Example 5.6 In equation [Tc6] in Figure 5.1 the function lookup is applied to *Tenv*. If this equation is applied during reduction of some term tcp(Program), a type-environment will appear as an auxiliary term for the attribute tcstm of an assign tree. Moreover, this type-environment is the value of the inherited attribute inh(tcstm,1). This attribute is a copy attribute. □

5.5.3 Multiply attributed term with function cache

Let the Φ_2-term $\xi_2(Aux, t_1, \ldots, t_m)$ occur in the reduction of $\phi(T)$, then Aux is decorated with an attribute graph for ξ_2. If Aux is the value of a copy attribute, it is possible that during the same reduction (of $\phi(T)$) also a term $\xi_2(Aux, t'_1, \ldots, t'_m)$ ($t'_j \neq t_j$) occurs. The attributes for ξ_2 of Aux already contain a value, however these values are based on the previous parameters $t_1, \ldots, t_m$. Therefore we decorate Aux with an extra attribute graph for ξ_2. During reduction of $\xi_2(Aux, t'_1, \ldots, t'_m)$ values are stored in attributes of this new graph. The result is that Aux is a *multiply attributed term*.

A *multiply attributed term* is a term which may have any number of attribute graphs for the same incremental function. The values of the inherited attributes at the beginning of each graph, that is, at the top node of the term, differ. To distinguish attributes of different graphs we label them with the values of the inherited attributes at the top.

Synthesized attributes without corresponding inherited attributes are not duplicated for multiple attribute graphs. They can safely be used in all graphs because the values of these attributes is not determined by the value of inherited attributes at the top.

A *hash table* is connected to a multiply attributed tree to allow for fast searching in the list of synthesized attributes at the top. The hash keys are calculated from the name of a function and the values of its parameters. The entries are (pointers to) the corresponding synthesized attributes. In this way we create a *function cache* for operations on the auxiliary term.

Example 5.7 In a program with several identifiers in the statements the one auxiliary term type-environment is used for the lookup of all these identifiers. So, this type-environment is decorated with a different Id-lookup graph for each identifier looked for.

Figure 5.7 shows a multiply attributed type-environment connected to a program term with type check attributes. The multiply attributed type-environment is the value shared by the tcdecls attribute and its copy successors. It is also the auxiliary tree for both the synthesized attribute tcstm at the tree assign(X,Exp1) and the tcstm attribute at the tree assign(Y,Exp2). The type-environment is attributed with two lookup-graphs. The function cache for Tenv has entries for the results of lookup X and lookup Y. The functions id-of and type-of have no parameters, so their associated attributes are shared by the two attribute graphs. □

5.6 Fine-grain incremental evaluation

We are now ready to explain fine-grain incremental evaluation, with multiply attributed terms connected to an attributed term. Function caches and definition-use dependencies are maintained and updated dynamically.

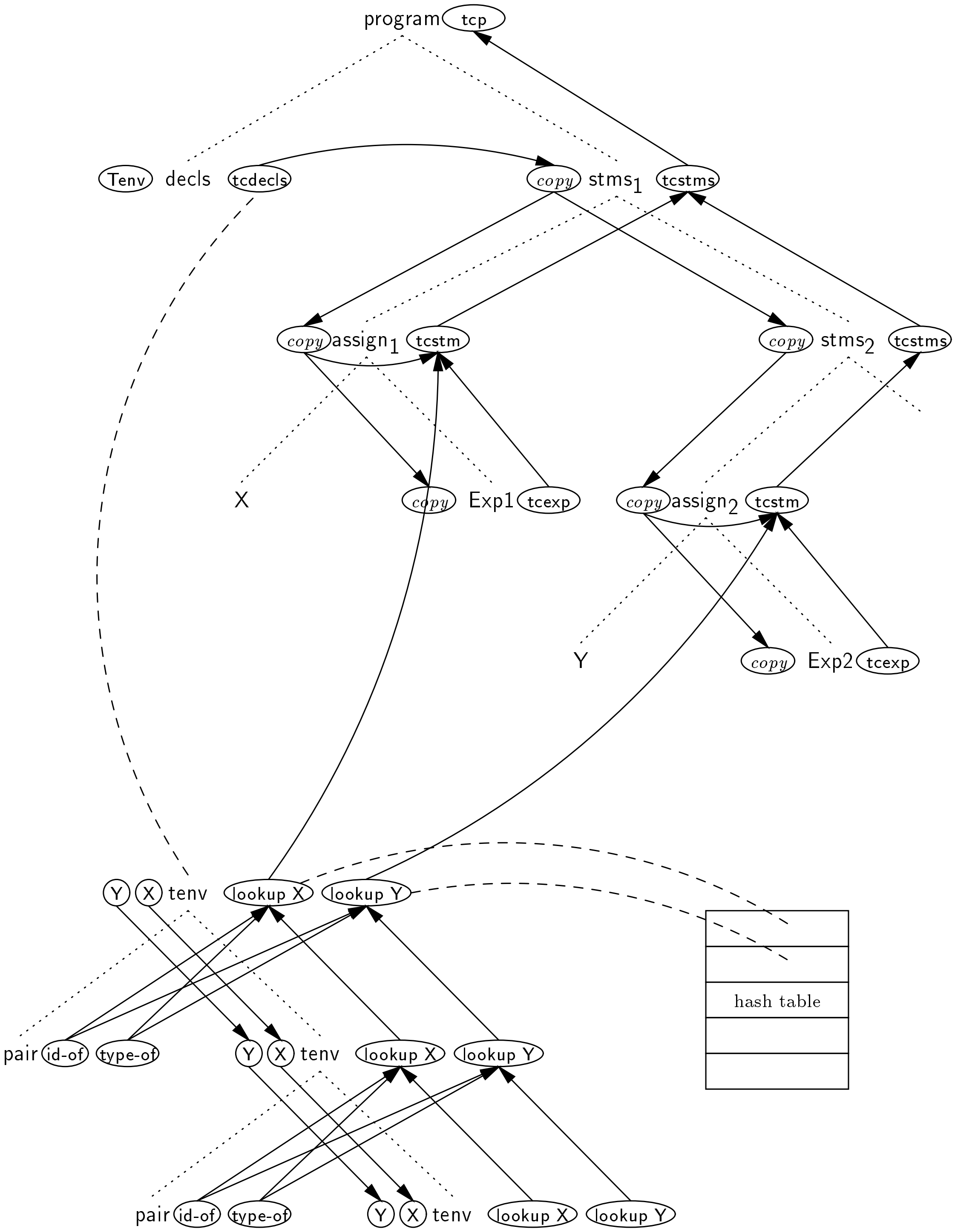


Figure 5.7: Attributed abstract syntax tree of a program. The value of the tcdecls attribute is a multiply attributed term

5.6.1 Initial reduction

Let T be a G_1 term and the attributed abstract syntax tree of a text in an editor. An operation ϕ on T is performed so the term $\phi(T)$ is reduced.

Assume that during the reduction of $\phi(T)$ a term $\phi_2(Aux, par_1, \ldots, par_m)$ occurs and Aux is the value of an attribute. We check the function cache related to this attribute value.

If an entry for ϕ_2 with parameter values $par_1, \ldots, par_m$ does not exist, then Aux is decorated with a new attribute graph for these parameters, and a key is added to the hash table together with (a pointer to) the new synthesized attribute at the top. The Φ_2-term is reduced and meanwhile the attributes in the new graph obtain a value.

If, on the other hand, the cache for Aux does contain an entry for ϕ_2 with parameter values $par_1 \ldots par_m$, then the synthesized attribute in the entry contains the normal form of the Φ_2-term and reduction can be avoided.

At each visit to a function cache, dependencies are established between the synthesized attribute in the entry and the attribute that uses its information. This edge *replaces* the edge from the *copy-head* attribute containing Aux to this *use*-attribute.

Example 5.8 We return to the specification in Figure 5.1 and the picture in Figure 5.7. During reduction of tcp(Program) equation [Tc6] is applied to reduce tcstm(assign(X,Exp),Tenv), which means that lookup(Tenv,X) must be reduced. If the function cache for Tenv contains no entry for lookup X, Tenv is decorated with attributes for lookup X. While lookup(Tenv,X) is being reduced intermediate results are stored in these attributes. A hash key for lookup X is added to the hash table, together with the synthesized lookup X attribute at the top of Tenv. The remote dependency between the attributes tcdecls and tcstm that have been created at edit time is replaced by the dependency between the synthesized attribute for lookup X and tcstm. □

For the specification of the type checker in our running example fine-grain incremental rewriting gives rise to memoization of the lookup function: During the reduction of tcp(Program), a term lookup(Tenv,Id) is reduced at most once for each Id in the statements section of the program. In Chapter 7 the performance of coarse-grain and fine-grain incremental rewriting is evaluated. The Figures 7.5 and 7.6 clearly show the memoizing effect for fine-grain incremental evaluation.

5.6.2 Updating attributes

If after the initial evaluation of $\phi(T)$ the text in the editor is modified so that subtrees are replaced in T, updating of attributes in T takes place as described in Section 5.3 for the standard case with copy dependencies.

Subsequent reduction of $\phi(T')$ with T' the modified term, is similar to standard incremental reduction except when an attribute obtains a new value and the previous value is a multiply attributed term. In that case the difference between the old value $OldAux$ and the new value $NewAux$ is computed and subtrees are replaced in $OldAux$ to create $NewAux$. The algorithm PROPAGATE-TE-UP of Figure 2.11 is applied to the synthesized attributes at the nodes where subtrees have been replaced. In this way all upward and horizontal successors of these attributes in *NewAux* are marked as unreliable, and so are their successor attributes in T'.

Later in the reduction some term $\xi_2(NewAux, \ldots)$ is likely to occur. The resulting normal form will be stored in the associated ξ_2-attribute at the top of *NewAux*. Remember that the entries in the function cache for *NewAux* are pointers to Φ_2-attributes. The function cache is updated incrementally during reduction simply by storing new attribute values.

It may happen, of course, that the new value of the ξ_2-attribute is equal to the old value. The algorithm PROPAGATE-UNCHANGED of Figure 2.13 then makes up for the cases in which initial propagation by PROPAGATE-TE-UP has marked too many attributes in *NewAux* and T', by resetting the status of these successors to "Unchanged".

Example 5.9 If in the program of Figure 5.7 the type of the identifier X is modified during editing, then during reduction of tcp(Program) a new value for tcdecls will be computed. We assume that the new type-environment can be obtained from the previous attribute value by replacing the Pair containing X. The status of the two attributes id-of and type-of of this new pair will become "Changed". The status of their upward successors in the type-environment (the lookup attributes) as well as the successor in the program tree (tcstm and tcstms attributes) will become "TobeEvaluated".

The first time the term lookup(Tenv,X) occurs in the reduction its new value will be computed and stored in the synthesized attribute. Every other occurrence of lookup(Tenv,X) will find this new value in the function cache.

The first time the term lookup(Tenv,Y) occurs in the reduction it will be detected that this value has remained the same, so PROPAGATE-UNCHANGED is invoked. □

Example 5.10 (Block structure) If the specification in Figure 5.1 would contain a block construct, a typical equation to specify its type checking would be

$$[\mathsf{Tc7}]\ \mathsf{tcstm}(\mathsf{block}(\mathit{Decls},\mathit{Stms}),\mathit{Tenv}) = \mathsf{tcstms}(\mathit{Stms},\mathsf{add}(\mathsf{tcdecls}(\mathit{Decls},\mathsf{empty\text{-}env}),\mathit{Tenv}))$$

where add is a function for the composition of type-environments. Since lookup(Tenv, Id) returns the type of the first occurrence of Id in Tenv, the declarations in the inner block take precedence over those in the outer block.

The type-environment attributes at the statements in a block are copies of one copy-head attribute. Each block has thus its own attributed Tenv, and its own hash table for lookup values. This copy-head is not the tcdecls attribute, as in Figure 5.7, but its successor, the inherited attribute for tcstms. *Use* attributes of a copy-head are both the tcstm attributes of statements in the same block and the copy-head attributes in the nested blocks.

When a declaration in a block is altered, the copy-head type-environment changes and its *use* successors, including the type-environments in the nested blocks, are marked as unreliable. During reduction, the new value of the type-environments in the inner blocks will be computed and their successor attributes will be marked and updated, exactly as described for the outer block. □

5.7 Discussion

The advantages of the method for fine-grain incremental rewriting is that it is automatically generated for layered PRSs, and that it requires no extension of the specification formalism. Moreover, in order to implement fine-grain incremental rewriting only little extra code has to be added to the code for coarse-grain incremental rewriting.

The principle idea of incremental rewriting being leftmost-innermost rewriting with short cuts and side effects for updating attributes is still valid. Consequently, the evaluation of attributes is demand driven and the status updating of attributes is data driven. In our type check example this means that after a small modification in the declarations the following type check attributes in the statements section will be re-evaluated:

$\mathit{Affected}_{fg}$, the attributes whose value depend on the modified declaration;

$\mathit{path\_to\_\ roots}_{\mathit{Affected}_{fg}}$, the upward successors of these attributes;

First, the set of attributes of statements or expressions with an identifier that does not occur in an earlier statement or expression, and

$\mathit{path\_to\_roots}_{\mathit{First}}$, the upward successors of these attributes.

All type check attributes in the statements section are visited by the status-propagation algorithms. Fortunately, resetting the status of attributes is a very simple operation.

The method is effective in the first place because of the memoization of lookup values. In the second place because after has been determined for an identifier that its lookup value has not changed, attributes which depend on that value can be marked as “Unchanged”.

5.8 Related work

5.8.1 Attribute grammars

Unlike algebraic specifications, attribute grammars maintain a strict separation between the domains of syntax description and semantics description. In attempts to solve the problem of aggregate values in attribute grammars either a predefined data type is added to the specification formalism or the attribute grammar formalism is extended or combined with another formalism.

In [HT86a] Hoover and Teitelbaum describe how symbol tables can be dealt with efficiently in the Synthesizer Generator [RT89a], an attribute grammar based system for specifying languages. A special class of data types, called finite functions data types, is added to the specification language. A 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. Direct definition-use links are established between components in the symbol table and attributes depending on that component. Difference propagation is used to determine the portion of the aggregate value that has changed.

In [RMT86] the specification language is also extended with a mechanism for defining tables and operations on tables. A special relation is defined between attributes whose values are defined by means of predefined table operations like creating, updating and looking up. This too results in direct definition-use links between attributes in the tree.

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.

5.8.2 Higher-order attribute grammars

Higher-order attribute grammars (HAGs) have been designed to remove the separation between the syntax description and the semantic description in attribute grammars [VSK89, TC90]. In implementing higher-order attribute grammars, Vogt et al. [VSK90, Vog93] used, as we did, the principle of applying incremental methods for operations on programs and their substructures to operations on attributes. In describing a symbol table by means of grammar rules with attributes for lookup values, an incremental implementation of the lookup function is obtained.

An incremental implementation for ordered higher-order attribute grammars is based on caching the results of visit functions for evaluating attributes [Kas80], rather than on storing attribute values in a tree. A visit function takes as first parameter a tree and part of the inherited attributes of the root of that tree. It returns a subset of synthesized attributes.

An immediate consequence of this is a memoization of attribute values at physically different trees with identical structure. In particular, the visiting results for a type-environment are cached and reused upon later visits to a copy of it.

In their approach *all* trees with identical structure are shared. The memoization effect of the method is therefore stronger than of our method. Especially for block structured languages this is an advantage.

If after a modification in the declarations the resulting type-environment has changed the complete statements section has to be revisited. As a result of sharing identical trees the unchanged part of the type-environment is maintained. So, when during the computation of the new type-check values of the statements an identifier has to be looked up in the new type-environment the cached results for the unmodifed part of the type-environment can be re-used. Clearly the lookup information in the new environment has to be computed once for each identifier.

Since attribute dependencies do not exist there is no way to indentify parts of the tree that need not be visited if the lookup value of an identifier turns out to be unchanged. All attributes in the statements section must be evaluated.

In Appendix B we present a translation of a *layered* PRS into a strongly non-circular HAG, as well as the translation of a strongly non-circular HAG into an algebraic specification which is not necessarily a *layered* PRS. Each well-presented PRS is equivalent to a strongly non-circular attribute grammar and vice versa. It is only logical that not every strongly non-circular HAG can be translated into a *layered* PRS, since *layered* PRSs are a subclass of well-presented PRSs, whereas strongly non-circular HAGs are a superclass of strongly non-circular attribute grammars.

5.9 Conclusions

For algebraic specifications in the class of layered primitive recursive schemes a fine-grain incremental implementation can be derived automatically without extending the specification formalism and with only small extensions to the algorithms for coarse-grain incremental rewriting.

For incremental functions that do not apply to a substructure of an edit tree this implementation creates an auxiliary attributed tree in addition to the attributed edit tree. This is profitable when equal auxiliary trees are used in several places, which is likely in the case where auxiliary trees are the values of *copy* attributes.

With each auxiliary tree a function cache is associated containing the synthesized attributes at the top of this tree. The values of these attributes are the results of applying an incremental function to the tree.

The method provides an efficient implementation of functions like the lookup operation on symbol tables in a type checker. Thus it repairs the common shortcoming that the basic, coarse-grain, incremental implementation presents when dealing with aggregate values.

We did not pay any attention to the *construction* of aggregate values. Exploration of constructor dependencies, as a generalization of copy dependencies, will probably lead to the incremental construction and updating of aggregate values. The components of a aggregate value are usually constructed from the values in predecessor attributes. If dependencies link these values to the exact component in the aggregate value, a change in a declaration will yield the modification in the generated aggregate value, without extra calculations. The explicit calculation of the tree difference in auxiliary terms can then be avoided.

Moreover, in a block structured program constructor dependencies would provide an opportunity to share part of the type-environment of a nested block with the type-environment of its surrounding block.

Chapter 6

An implementation of incremental rewriting

> The incremental techniques described in the previous chapters of this thesis have been implemented in the ASF+SDF meta-environment. The rewrite engine of the system implements a leftmost innermost strategy, and provides facilities for adding alternative strategies. Incremental rewriting is thus implemented as an extension to the standard rewrite engine.

6.1 Introduction

We have implemented incremental rewriting as an extension of the rewrite engine of the ASF+SDF meta-environment [Kli91]. The ASF+SDF meta-environment is a programming environment generator based on algebraic specifications. From a language specification written in the ASF+SDF formalism, an environment is generated. The main components of this environment are a scanner and a parser for programs in the specified language, a syntax directed editor, controlled by the scanner and the parser, and a term rewrite system. Buttons can be added to the editor, which when pressed activate operations, like type checking, compilation, evaluation on the program in the editor. The term-rewrite system implements a leftmost innermost reduction strategy but is *extendible* so that alternative rewrite strategies can be added easily. Incremental rewriting as described in Chapters 2–5, is one of these alternative strategies.

In order to obtain an incremental implementation of an ASF+SDF specification, the keyword incremental can be added to the declaration of operators of which the first argument is the sort of a term that may occur in the editor, or the sort of a subterm of an edit term, typically sorts that occur in the syntax specification of a language. Such an operator is then called an *incremental operator*, and we refer to the sort of its first arguments as *grammar sort*. Other arguments of an incremental operator are called *parameters*.

A specification with incremental operators must meet certain requirements. Let Φ be the signature of the incremental operators with Eq_Φ the defining equations, let G be a signature formed by the grammar sorts with their constructors, and let S be the

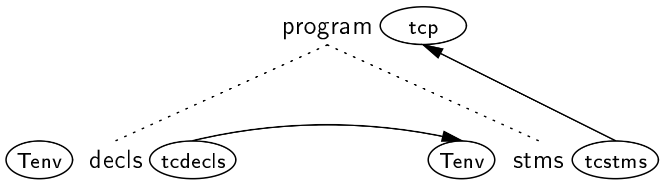


Figure 6.1: Top of a parse tree, with attributes for type check operators

remaining signature, with Eq the remaining equations. The complete specification, $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$, must then be a PRS or a layered PRS, according to the description in the previous chapters.

We have implemented incremental rewriting for terms like $\phi(T, t_1, \ldots, t_m)$, with T a term in the editor and ϕ an incremental operator. A consequence of the properties of a PRS is that the first subterm of each incremental term that appears in the reduction of $\phi(T, t_1, \ldots, t_m)$, is always a subterm of the edit term T. Normal forms of incremental terms and parameters are therefore stored in attributes of the parse tree. To this end, we associate with each incremental operator one synthesized attribute, and with each parameter of the operator we associate an inherited attribute. When an incremental term $\psi(U, u_1, \ldots, u_{m'})$ appears during rewriting, two situations may occur: reductions are performed and the resulting normal forms are stored in attributes of the subtree U of the parse tree, or attributes contain previously stored normal forms which are used to avoid rewriting steps.

Attributes are connected by means of *dependencies*, that are derived from defining equations. When the term T in the editor is altered into T', subtrees are replaced in the attributed parse tree of T. Attributes in the parse tree of T', that depend on the attributes in the old subtrees are marked as unreliable. During subsequent reduction of $\phi(T', t_1, \ldots, t_m)$, reliable attributes are used to avoid rewrite steps, and unreliable attributes obtain a new, correct, value.

Example 6.1 Figure 6.1 shows the top of the parse tree of a program, with attributes for the incremental operators tcp, tcdecl and tcstms. These attributes obtain a value during reduction of a term tcp(program(Decls,Stms)). The depicted attribute dependencies are derived from the equation

$$\mathsf{tcp}(\mathsf{program}(Decls, Stms)) = \mathsf{tcstms}(Stms, \mathsf{tcdecls}(Decls, \mathsf{empty\text{-}tenv}))$$

□

The implementation can handle *multiple* subtree replacements, so that incremental re-computation can take place after any number of edit actions. The implementation caters for specifications with conditional equations, incremental operators over lists, and a varying granularity for incrementality.

The standard strategy provides an incremental implementation for operators over sorts that appear in an editor. For a further improvement of the performance, operators of which the sort of the first argument appears as a parameter sort of other incremental operators, can be declared to be incremental as well. This results in a fine-grain incremental implementation and can be applied for efficient implementation of operators over attribute values that are aggregates.

The implementation language is LeLisp [LeL90], a Lisp dialect. The basic programming unit in LeLisp is a *function*. To avoid confusion between LeLisp functions and functions in an ASF+SDF specification, we use *operator* in this chapter to indicate the latter kind of functions.

Overview of this chapter

In the remainder of this chapter we discuss the interfaces with the rewrite engine and the editor in a programming environment and some non-standard features of the implementation. Section 6.2 discusses the global set up of the extendible Equation Manager. Section 6.3 discusses the standard preprocessing of equations and the extensions for incremental reduction. In Section 6.4 we describe the standard reduction strategy as well as the handles for extending the rewrite engine with alternative strategies. The particular extension for incremental reduction is described in Section 6.5. The interface with the editor is described in Section 6.6. In Section 6.7 we highlight implementation aspects of non-standard features like syntactic references, varying granularity and fine-grain incrementality. Section 6.8 contains some conclusions.

6.2 The Equation Manager

The Equation Manager, EQM for short, is the piece of code in the ASF+SDF meta-environment that analyzes equations, generates the term rewrite system and performs the actual rewriting. The Equation Manager has been designed and written by C.H.S. Dik.

Whereas the standard reduce functions implement a leftmost innermost rewrite strategy, the Equation Manager has been designed in such a way that alternative rewrite procedures can be added easily and, to a large extent, independently. Its extendibility permits on the one hand easy experiments with techniques like lazy rewriting, outermost strategies, higher order rewriting, incremental rewriting, and debugging. On the other hand, it ensures clean interfaces between the core of the Equation Manager and the various extensions. In Section 6.3–6.5 a detailed description is given of the Equation Manager and the extensions for incremental rewriting. Here, we first give a global description.

The key idea of the extendible rewrite engine is that a term is reduced by applying the particular reduce function associated with the top-operator of the term. During preprocessing of equations, each strategy can declare the name of its reduce function, and select the operators that belong to it.

Extending the Equation Manager with a new strategy is done by simply declaring the names of *hook functions*. At certain designated points during preprocessing of equations, and at reduction time, control is given to alternative strategies by invoking one of their *hook functions*.

Data structures for preprocessing, and runtime data structures are extended with an extra field containing a table. Each strategy can add information to these tables, with a key that indicates the strategy itself. Only functions that belong to a certain strategy are supposed to access the information relevant to that strategy.

New strategies can thus be defined easily, yet not entirely independently. It may happen that different strategies select the same operator. The reduce function of the strategy whose *hook functions* are invoked last, is then associated with the operator in question.

6.3 Preprocessing

In the Equation Manager, the preprocessing of equations consists of three steps. First, each equation is checked and a data structure, EQstruct, is constructed. An EQstruct typically contains all parts of the equation such as, left-hand side, right-hand side, and conditions, as well as extra information, like error messages related to this equation. An extra field in EQstruct is reserved for storing information for alternative strategies. This field contains a table. Each strategy can add its own pair with a key to indicate the strategy and the information for that strategy as entry. For each alternative strategy a *hook function* is called, to perform additional checks on the equation and store extra information.

Second, all EQstructs are hashed according to the top operator of the left-hand side. For each alternative strategy and for each operator, a *hook function* is called with three arguments: the operator, the equations for that operator, and a structure with information on all operators and equations of the specification.

Third, a *hook function* is called for each strategy so that global checks can be performed and information relevant to all operators and all equations can be stored.

Hence, the designer of an alternative strategy has to define three *hook functions* for preprocessing. We discuss now the ones for incremental rewriting.

6.3.1 Preprocessing per equation

For incremental rewriting, we distinguish incremental equations- that is, equations of which the top-operator of the left-hand side has been declared as incremental- and non-incremental equations.

For each non-incremental equation, we check that no incremental operator occurs anywhere in the equation. For each incremental equation we check that the requirements for a well-presentable primitive recursive scheme, as formulated in Chapter 2 are met. For incremental operators over list sorts, checks are performed as described in Chapter 4. If equations are not correct, warnings or error messages are generated, with suggestions for reparation.

Information for incremental reduction is stored in the EQstruct for each equation. For the right-hand side of the equation, *attribute dependencies* are derived for the right-hand side of the equation, and for the terms on both sides of each condition as described in Chapter 2 and Chapter 3. For incremental operators over lists, information about the attribute dependencies for lists is gathered, as described in Chapter 4.

For each incremental term in the right-hand side and conditions, we store the *rank* (child number) of its grammar term relative to the grammar term in the left-hand side. For instance, in the equation

$$\mathsf{tcp}(\mathsf{program}(\mathit{Decls}, \mathit{Stms})) = \mathsf{tcstms}(\mathit{Stms}, \mathsf{tcdecls}(\mathit{Decls}, \mathsf{empty\text{-}tenv}))$$

the rank of the grammar term in $\mathsf{tcstms}(\mathit{Stms}, \mathsf{tcdecls}(\mathit{Decls}, \mathsf{empty\text{-}tenv}))$ is 2, i.e, the rank of *Stms* in $\mathsf{program}(\mathit{Decls,Stms})$.

If the grammar term is not the subterm of the grammar term in the left-hand side, we require that it equals a parameter term in the left-hand side. In that case the rank information contains a description of the inherited attribute associated with the parameter. A fine-grain incremental implementation will be derived, as explained in Section 6.7.3.

If a grammar term occurs in the right-hand side or condition of an incremental equation, and this grammar term is not the subterm of an incremental term, a *syntactic reference* attribute is associated with this grammar term. In Section 6.7.1 we discuss the subject of syntactic references.

6.3.2 Preprocessing per operator

In the second phase of preprocessing, we determine for each incremental operator its *grammar sort*, i.e. the sort of its first argument.

6.3.3 Global checks

In the third phase, we check whether the *set* of equations for incremental functions is well-presentable.

The name of the function for incremental reduction is stored for each incremental operator. Extra arguments for this function are needed for the creation of attributes and attribute dependencies during incremental reduction. These arguments are the list of all *grammar sorts* and the dependency relation for attributes of grammar sorts that are lists. For incremental equations with a list grammar sort, information on the dependency relation per equation is combined, to yield the dependency relation of attributes of the list sort.

6.4 An extendible rewrite engine

As mentioned before, the key concept of the rewrite engine of the Equation Manager is that a term is reduced by applying the function associated with the top-operator of the term. During preprocessing of equations, the name of a reduce function is associated with each operator. In this section we explain the default strategy together with its handles for allowing combination with alternative strategies.

Figure 6.2 presents an overview of the internal structure of the rewrite machine, and shows both the default rewrite strategy and the extension for incremental rewriting.

The double box with `ReduceTree`, on top of the figure, together with the diamond shape, `op?`, express that when a term is to be reduced the `ReduceTree` function checks the top operator and chooses whether the function `LeftmostInnermost` is to be used or the function `IncReduce`.

6.4.1 Leftmost innermost reduction

We explain the leftmost tree of Figure 6.2, while ignoring the ellipses with `CondInfo` and `RhsInfo`.

`ReduceSon`: When reducing a term according to the leftmost-innermost strategy the subterms of the term are reduced first by recursively applying the `ReduceTree` function to these subterms. The subterms are replaced by their normal forms.

`ReduceOnce`: Next, the function `ReduceOnce` is applied to the term. The function `FindEq` tries, by means of `CheckEq`, to find an equation of which the left-hand side matches the term and the conditions succeed.

During matching, `Match`, a substitution table of variable and terms is built. If matching succeeds and conditions are checked by `CheckCond`, this table is used to substitute the variables in the condition in `SubsLeft` and `SubsRight`. After substitution, each side of the condition is reduced to normal form by applying the function `ReduceTree`. The function `Compare` compares these two normal forms. A condition succeeds when they are equal.

When an equation has been found with a matching left-hand side and succeeding conditions, the term in its right-hand side is instantiated using `SubsRhs`. Finally, the original term is replaced by this instantiated right-hand side.

Reduction stops when `CheckEq` does not find an applicable equation. The term is then marked as normalized.

`ReduceTree`: If the term is not marked as normalized the function ReduceTree is called again to further reduce the term.

6.4.2 Handles for alternative strategies

Three small extensions to the basic rewrite machine were needed to make it possible to use alternative rewrite strategies in combination with the standard reduce functions.

First, an extra argument is added to all reduce functions, in which information could be maintained for alternative strategies. The argument is a table, in which a key indicates an alternative strategy and the related table entry indicates the runtime data structure for that strategy.

Second, calls to *hook functions* of alternative strategies are done by `CondInfo`, before a term on a condition side is reduced, and by `RhsInfo` after the right-hand side of an equation is instantiated, that is, before reduction proceeds with the instantiated right-hand side. The *hook functions* can update their runtime data structures with information from the equation at hand.

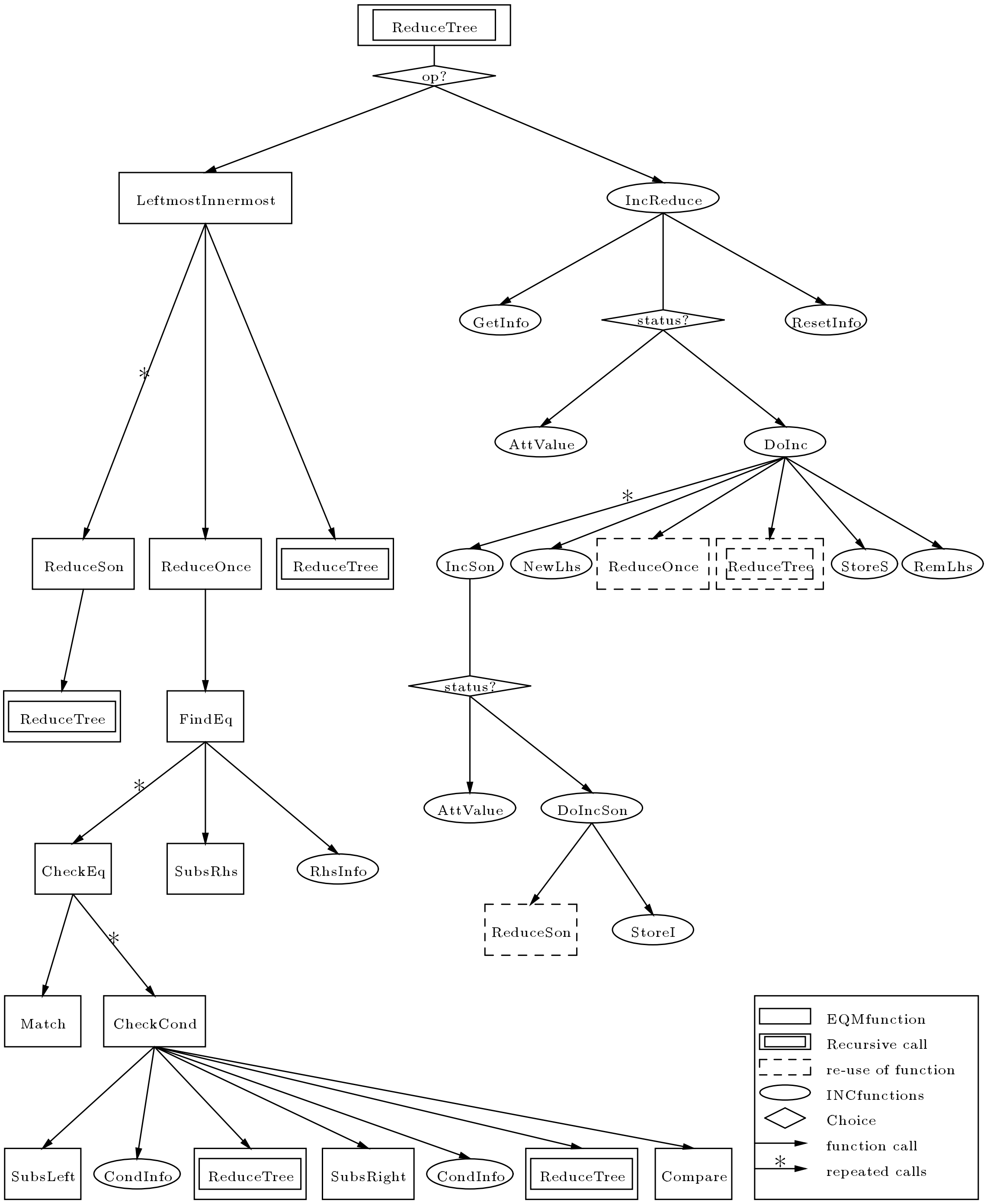


Figure 6.2: Representation of the internal structure of the rewrite engine of the ASF+SDF meta-environment. The tree on the left contains the functions implementing a leftmost innermost rewrite strategy. The ellipses with `CondInfo` and `RhsInfo` invoke *hook functions* for alternative strategies. The tree on the right contains the functions for incremental rewriting.

Finally, to ensure transparency and maintainability, macros for invoking default reduce functions or alternative reduce functions are provided. When a reduce function is called by means of a macro, standard arguments with standard names are passed along. The designer of an alternative reduce function only has to know the names for relevant arguments, usually only the term to be reduced, and its top-operator. When, for some reason, the number of arguments for reduce functions is changed, only these macros have to be updated.

Three *hook functions* must be declared to extend the rewrite engine with an alternative strategy: the actual reduce function, a function to be invoked by `CondInfo`, and one that is invoked by `RhsInfo`.

6.5 Extending the rewrite engine

Incremental reduction has been implemented by extending the rewrite engine with functions for reducing terms with an incremental operator. We first discuss the use of storage in incremental rewriting. Next, we treat the functions invoked by `CondInfo` and `RhsInfo` for updating this data structures with information stored in equations. Finally, we explain the incremental reduce functions.

6.5.1 Storage for incremental reduction

Our concern is incremental rewriting of an initial term $\phi(T, t_1, \ldots, t_m)$, of which the grammar term, T, is the term in the editor. Intermediate normal forms are stored in attributes of the parse tree of T.

Attributes and dependencies

Attributes have a *name*, a *value*, a *status*, and a *graph-id*. The *name* of a synthesized attribute for an operator ϕ is the pair $(\phi, 0)$. The name of an inherited attribute associated with the k-th parameter of ϕ is the pair (ϕ, k). An attribute *value* is a normal form. The *status* is meant to indicate whether a value can safely be used or not. For efficiency reasons explained in Chapter 2, we have two status indications for reliable attributes: "Unchanged" and "Changed", and two status indications that indicate a value is not reliable: "TobeEvaluated" and "Initial". The *graph-id* is used to distinguish between attributes of different graphs in a multiply attributed tree as used by fine-grain incrementality (Section 6.7.3).

Attributes are connected by means of a dependency graph. When T is edited, attributes at the top of a new subtree get status "Initial". Other attributes whose value may depend on the modification, are marked as "TobeEvaluated". When a new normal form is stored in an attribute and this new value equals the previous one, its status becomes "Unchanged"; otherwise it is set to "Changed". The status of "Changed" attributes is reset to "Unchanged" when reduction is finished.

Attributes and dependencies are created during reduction. Dependencies are updated dynamically for efficiency reasons as discussed in Chapter 3. Incoming edges of synthesized attributes are therefore labeled with the equation that has introduced them.

Runtime data structure

The runtime data structure contains dynamic information and static information. The dynamic parts consists of pointers to attributes and to subtrees of the parse tree:

- A pointer to the attributed parse tree.
- A stack of pointers to *current tree*, a subtree or sublist of the attributed tree that corresponds to the first subterm/the grammar term of the incremental term being reduced.
- A stack of pointers to the *left-hand-side tree*, this is also a subtree or sublist of the attributed tree, usually the parent tree of current tree. During incremental rewriting, subtrees of the attributed tree are visited in an order that is induced by the equations that are applied. When an incremental term occurs in the instantiated right-hand side or condition of an equation, its grammar term is always a subterm of the grammar term of the term matching the left-hand side of this equation. Therefore, we keep a reference to this left-hand side tree.
- A stack of pointers to *current attributes*, attributes of *current tree* that are associated with the incremental operator of the term being reduced.
- A list of *changed attributes*, attributes whose value has changed and whose status has been set to "Changed".

The static information used at runtime is information created during preprocessing. It consists of global information needed for creation of attribute and dependencies, which is not stored in EQstructs.

- The *attribute dependencies* for incremental operators over lists, needed for creation of attribute dependencies.
- A list of *grammar sorts* needed for the dynamic creation of *syntactic reference* attributes, (Section 6.7.1).

6.5.2 `CondInfo` and `RhsInfo`

During preprocessing, information about attribute dependencies for terms in the conditions and the right-hand side term of an equation, as well as rank information for the incremental subterms, is stored in the EQstruct of that equation.

The *hook functions* that are invoked by `CondInfo` and `RhsInfo` use this information to create and update attributes and dependencies for the condition term or right-hand side term to be reduced. Moreover, they store rank information in the relevant incremental subterms so that the proper rank of an incremental subterm will be known, regardless of the order in which these subterms are to be reduced by `IncReduce`. (Another solution that would achieve the same effect would have been to store a pointer to each incremental subterm together with its rank.)

6.5.3 Incremental reduction

We are now ready to explain the rightmost part of the tree in Figure 6.2. The function `IncReduce` is applied to reduce terms with an incremental top-operator. First, we explain the three steps at the top.

`GetInfo`: When incremental reduction of a term starts, the runtime data structure is updated. The *left-hand-side tree* is used to find the new *current tree*:

If there is no *left-hand-side tree*, the first subterm of the incremental term must equal the stored attributed term. The pointer to the top of the stored tree is put on the stack for the *current tree*.

Otherwise, the rank information stored in the incremental term is used to find the *current tree* as a subtree of *left-hand-side tree*.

If the grammar term of the incremental term is a sublist or a list element, we only know its rank with respect to the surrounding sublist. The *left-hand-side tree* is then a sublist; we use the rank of the first element of the *left-hand-side tree* to compute the absolute rank of the *current* element or sublist.

When the rank information contains an attribute indication, then the new *current tree* is the value of an attribute of *left-hand-side tree*.

Among the attributes of *current tree* the attributes for the top operator of term are selected or created, if they are not present. A pointer to these attributes is put on the stack for *current attributes*.

`Status?`: If the status of the current synthesized attribute indicates that the attribute contains a reliable value, reduction can be avoided and the function `AttValue` is used to replace the term by the attribute value. Otherwise, the normal form of the term must be computed using `DoInc`.

`ResetInfo`: The pointer to the current tree is removed from the stack. When the stack of current trees is empty, incremental reduction is finished. The status of all *changed attributes* is set to "Unchanged", and the status of their "TobeEvaluated" successors is set to "Initial".

The function `DoInc` interleaves calls to `reduce` functions with calls to functions for retrieving information, storing information, and attribute updating.

`IncSon`: First, each subterm is replaced by its normal form; this is done incrementally by `IncSon`. When a reliable attribute status is found for the *current* inherited attribute associated with a subterm, reduction is avoided and `AttValue` replaces the subterm by this value.

Otherwise, `DoIncSon` calls the function `ReduceSon` to find the normal form of the subterm. The function `StoreI` stores the normal form in the proper *current* inherited attribute. The new attribute value is compared with the previous one. When the new value differs from the previous one, the attribute status becomes "Changed", and the attribute is added to the list of *changed attributes*. Otherwise, the status is set to "Unchanged". Status information is propagated to successor attributes. We discuss this in the next section. In Section 6.7.3 we explain what happens when an attributed attribute value changes.

`NewLhs`: Before reduction proceeds, the pointer to *current tree* is put on the stack of *left-hand-side trees*. This is done because the incremental term of which current tree is the grammar term, which is going to be matched to the left-hand side of an equation.

`ReduceOnce`, `ReduceTree`: The EQMfunctions `ReduceOnce` and `ReduceTree` are applied as was done for standard leftmost-innermost reduction.

`StoreS`: The normal form is stored in the *current* synthesized attribute. The rest of the procedure is similar to `StoreI`.

`RemLhs`: Finally, the pointer to the *left-hand-side tree* is removed from its stack.

6.5.4 Propagation functions

Propagation functions are functions that take an attribute in a dependency graph and propagate status information in successors of the attribute. For instance, when a new attribute value is stored in an attribute and this value equals the old one, the status of the attribute becomes "Unchanged". A propagation function then inspects the successors with status "TobeEvaluated". If a successor is found to have only "Unchanged" predecessors, its status is reset to "Unchanged" as well and propagation proceeds from there. Propagation functions are implementations of the algorithms in Chapter 2, and the algorithm in Chapter 5 for copy dependencies and remote successors.

To keep Figure 6.2 uncluttered we ignored the need for code of status propagation. It is possible that in `StoreI` the value of an inherited attribute has not changed, and that, as a consequence, the status of the corresponding synthesized attribute is set to "Unchanged". In that case further reduction of the term can be skipped, and the functions `NewLhs`, `ReduceOnce`, `ReduceTree` and `RemLhs` need not be applied.

6.6 Interface with the editor

In the ASF+SDF meta-environment, syntax directed editors [Koo92b] are generated for programs in a specified language. A syntax directed editor uses an incremental parse technique that is regulated by a focus. A focus is positioned around a part of the text in the editor corresponding to a subtree of the parse tree. Text outside the focus is always correctly parsed. Within the focus one can edit text. Upon request, the text within the focus is parsed. Buttons can be added to the editor, that, when pressed activate operations, such as type checking, compilation, and evaluation, on the program in the editor. More sophisticated facilities for customizing editors [Koo92a] can be used to declare operations as so called *active tools*. Instead of being invoked only on explicit request by the user, an active operation is applied automatically after each parse.

For incremental rewriting, information is stored in attributes of the parse tree of an edit term. When the term is edited and re-parsed, some information is discarded. We now discuss how the remaining information is updated.

After each parse, the top of a new subtree, i.e, the smallest subtree of a *grammar sort* that contains the modifications, inherits the attributes of the top node of the previous subtree. Hardly any attributes have to be created for the lower nodes of the new subtree, because attributes and dependencies can be created at run time. An exception is discussed in Section 6.7.1. Dependencies between attributes in the old subtree and attributes at the top of new subtree, as well as remote dependencies between attributes in the old subtree and attributes in the rest of the tree, are removed. If the new subtree is a list element that has been inserted, attribute dependencies are restored according to the dependencies derived from defining equations for incremental operators over lists.

The status of the synthesized attributes at a new subtree is set to “Initial”, and successors along a dependency path to the top of the parse tree are marked as “Tobe-Evaluated”, as described in Chapter 2.

6.7 Non-standard Features

We discuss in this section some non-standard features and the effects on the implementation. In real-life PRS-like specifications, many sorts used in the specification of the syntax of a language are also used in the description of its semantics. We introduce syntactic-reference attributes to handle this. Next, we discuss incremental implementation with varying granularity. Finally, we briefly mention some implementation aspects of fine-grain incremental reduction.

6.7.1 Syntactic references

In the definition of a PRS $\langle G, S, \Phi, Eq, Eq_\Phi \rangle$, the sorts of S and G are assumed to be disjoint. However, in existing type check 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.

If the result of an incremental operator applied to some constructor depends *directly* on the value of one of its subterms, a *syntactic reference* attribute or *syntref* attribute is created at this subterm, together with a dependency from this *syntref* attribute to the attribute for the incremental operator in question.

Example 6.2 The following equation specifies the type checking of a declaration that consists of an identifier and a type.

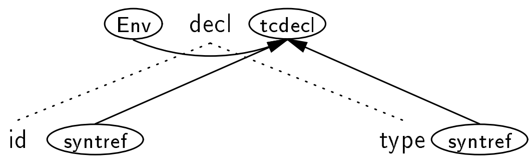


Figure 6.3: Syntref attributes

$$\mathsf{tcdecl}(\mathsf{decl}(Id, Type), Tenv) = \mathsf{add}(\mathsf{pair}(Id, Type), Tenv)$$

The result is a type-environment extended with an entry for that same identifier and its type. Thus, the value of the attribute for tcdecl depends directly on the subtrees *Id* and *Type*. Figure 6.3 illustrates how *syntref* attributes are applied to yield the proper dependencies. □

A *syntref* attribute is a synthesized attribute. Its *name* is *(syntref,0)*, and its value is the tree to which it has been attached. At most one *syntref* attribute is needed per subtree. When a *syntref* attribute is created for the top node of a tree T during reduction or at edit time, the attribute dependency graph D_T is extended further by adding a *syntref* attribute to all subtrees of T that belong to a *grammar sort*. This extension is recursively applied to these sons. The *syntref* attribute at a node is always the successor of the *syntref* attribute of its children.

6.7.2 Granularity

Sometimes reduction of a term is so simple that it is not worthwile to store the result for incremental evaluation. The granularity of a set of incremental operators can therefore be defined by the writer of a specification. One can, for instance, declare the type check operators for programs, declarations, lists of statements, and single statements, to be incremental, but omit the type check operators for expressions. In that case, the sort Exp does not belong to the *grammar sorts*.

Several groups of incremental operators with different granularity can be combined, e.g, type check operators with the above mentioned granularity, with compile operators that, for some reason, are incremental for expressions but not for declarations. The set of *grammar sorts* then includes both Exp and Decls. During preprocessing of equations, *syntref* attributes are associated with grammar terms that are not subterms of an incremental term. As a consequence, *syntref* attributes are automatically associated with terms of sort Exp during preprocessing of equations for type checking, and with terms of sort Decl during preprocessing of equations for compilation. In this way the grain size of the two sets of operators becomes identical.

For updating attributes after editing a subtree, we take as a starting point the smallest surrounding subtree that belongs to a *grammar sort*.

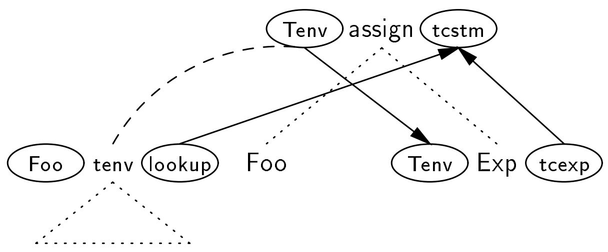


Figure 6.4: Attributed attribute value

6.7.3 Fine-grain incrementality

The standard strategy provides an incremental implementation for operators over sorts that appear in an editor. For a further improvement of the performance, operators of which the sort of the first argument appears as a parameter sort of other incremental operators, can be declared incremental as well. This results in a fine-grain incremental implementation, and can be applied for efficient implementation of functions over attribute values that are aggregates. In Chapter 5 we have described fine-grain incrementality. The actual implementation follows that description. During preprocessing of the equations, information for fine-grain incrementality is gathered when the grammar term of an incremental term in a right-hand side or condition equals a parameter term of the left-hand side. During reduction, this will result in an attribution of the value of the inherited attribute associated with that parameter term.

Example 6.3 Assume that the operators tcstm and lookup have been declared incremental. In the equation below the first argument of the lookup-term is *Tenv*, which is the parameter of tcstm in the left-hand side. During reduction of some term tcstm(assign(Foo,Exp),Tenv), the value Tenv of the inherited attribute (tcstm,1) will be decorated with attributes for lookup.

$$\mathsf{tcstm}(\mathsf{assign}(Id, Exp), Tenv) = \mathsf{compatible}(\mathsf{lookup}(Tenv, Id), \mathsf{tcexp}(Exp, Tenv))$$

□

The attribution of an attribute value differs from attribution of the abstract syntax tree of a program in an editor in that several attribute graphs may occur for one incremental function with different parameter values. To distinguish between attributes of different graphs, attributes are extended with the notion of a *graph-id*. The *graph-id* is computed from the function name and the parameter values.

The successors of attributes at the top of an attributed attribute value *Aux* are attributes in another tree: either the abstract syntax tree of a program in an editor or another attributed attribute value, if *Aux* belongs to a subPRS of another subPRS. During reduction, the function `StoreI` is used to store a new value in an inherited attribute. If the new value differs from the old one and the old value is attributed, the difference Δ is computed and subtrees are replaced in the old value so that it equals the new one. Attributes at the top of new subtrees are marked “Initial” and propagation functions are applied to set the status of their successors to "TobeEvaluated". If the status of attributes at the top of the new attribute value become "TobeEvaluated", status propagation is continued at the successor of such attributes.

6.8 Conclusions

We have described the implementation of incremental rewriting for conditional primitive recursive schemes, based on an innermost rewrite strategy. The implementation can handle multiple subtree replacements, incremental functions over lists, arbitrary granularity, and fine-grain incrementality. The interface with the standard rewrite engine in the ASF+SDF system is simple and elegant, thanks to the facilities for adding alternative rewrite strategies provided by the Equation Manager, the piece of code responsible for generating the term rewrite system and performing the actual rewriting.

In the next chapter we asses the performance of this implementation

Although Figure 6.2 may suggest differently, the code for the incremental extension to the Equation Manager is quite large. This is due mainly to the elaborate analyses of equations and the code for creation and updating of attributes and dependencies. The number of lines of code for the Equation Manager (EQM) and the extension for incremental rewriting (INC) give a rough impression.

Preprocessing in EQM	1000	lines
Reduction in EQM	750	lines
Preprocessing for INC	2500	lines
Reduction for INC	2300	lines

In other words, the implementation of the extra functions for incremental rewriting is roughly 2.5 times as large as the original, non-incremental, implementation.

Chapter 7

Performance evaluation of incremental rewriting

7.1 Introduction

To evaluate the performance of the implementation of the incremental techniques described in the previous chapters, we measured the cost for type checking Pico programs on the basis of the Pico specification in Appendix C.

It is obvious from the theoretical description that incremental reduction reduces the *number of rewrite steps*. Since incrementality inevitably comes with time and space overhead for storing additional information, we also discuss the *time* gain of our implementation

It should be noted that the effectiveness of the method depends largely on the particular specification to which it is applied. For instance, when the type checking rules cause most language constructs in a program to depend on many other parts of a program, our technique is not appropriate for speeding up the process. We believe, however, that the Pico type checker is a simple example of a large class of type checking specifications with similar incremental behaviour (e.g, most Algol-like languages).

We present the measurements for type checking Pico programs incrementally. We discuss overhead costs for initial incremental type checking, and incremental checking after modifications of different sizes have been made to the program. Next, we illustrate the effect of fine-grain incremental reduction. Finally, we show how much extra memory is used for storing information for incremental reduction.

In our measurements, we concentrate on the effects of incremental rewriting. Therefore, we exclude the time spent on necessary but non-essential tasks:

- Determination of modifications. In our current implementation these are determined by computing the difference between the original tree and the modified tree. In Section 6.6 is explained how the difference can be determined with some help from the syntax directed editor.

- Conversion between term representations. The term representation during rewriting differs from the representation used for attributed trees, which is the one used in syntax directed editors. This requires occasional conversion from one representation to another.

All measurements were performed on a Silicon Graphics workstation, type Iris INDIGO, running under operating system IRIX Release 4.0.5F System V, with 24 Mbytes internal memory.

7.2 Basic Implementation

Figure 7.1 shows the number of rewrites steps for type checking type-correct programs of various lengths as well as the time spent. It presents graphs for non-incremental type checking (not-inc), incremental type checking of a program without using any preliminary information (init-inc), and incremental type checking after a small modification (inc). Figure 7.2 presents the same information for programs that are not type-correct.

7.2.1 Initial costs

Description of the test In the test specification the type check functions tcp, tcdecls, tcstms, tcstm, tcexp were declared incremental.

We performed measurements on programs with 10 declarations and various numbers of statements. In the first test (Figure 7.1) the statements are simple type correct assignments like Id1 := Id2, in which the type of both identifiers is natural. In the second test (Figure 7.2) the statements are incorrect assignments like Id1 := Id2, in which the type of Id1 is natural and the type of Id2 is string.

In all programs we modified the last statement by replacing one identifier by a new undeclared identifier. This modification changed the type check value of this statement to compatible(natural,lookup(empty-tenv,Newid). It also changed the type check result of the whole program.

Discussion Initial incremental rewriting is the same as non-incremental rewriting but with some side effects. So the number of rewrite steps is the same in both cases. The overhead time for initial incremental rewriting is the time needed for the creation of attributes and their dependencies and storing normal forms in them.

From the difference between the graphs for non-incremental reduction and for initial incremental reduction we can derive that the overhead for initial incremental reduction for 140 assignments is approximately 1 second. Apart from the two attributes for tcp and tcdecls, a program with N assignments is decorated with $6N$ attributes. Hence, about 800 attributes can be created in 1 second.

The number of rewrite steps for incrementally processing a small modification is low because the type checking of all other statements can be avoided. There is overhead time for attribute updating. This includes time for status propagation, the costs for looking up attributes, and the costs for comparing attribute values.

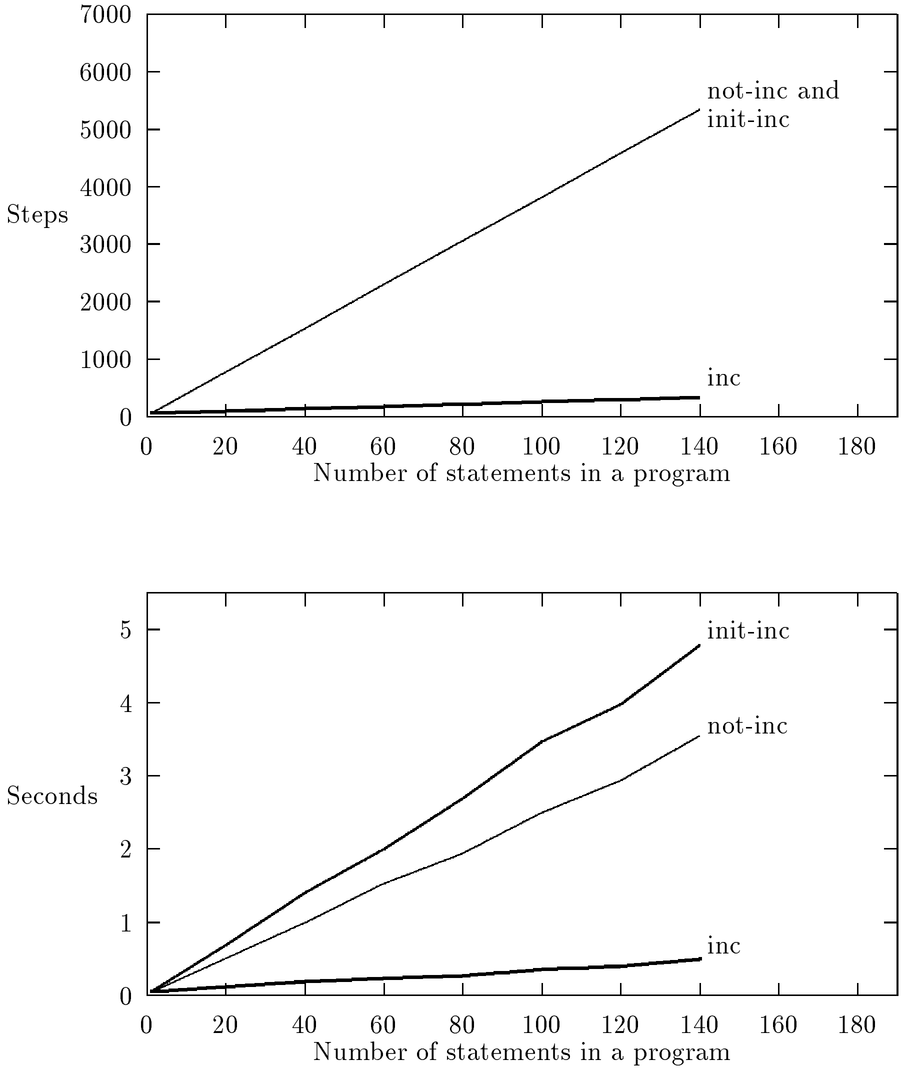
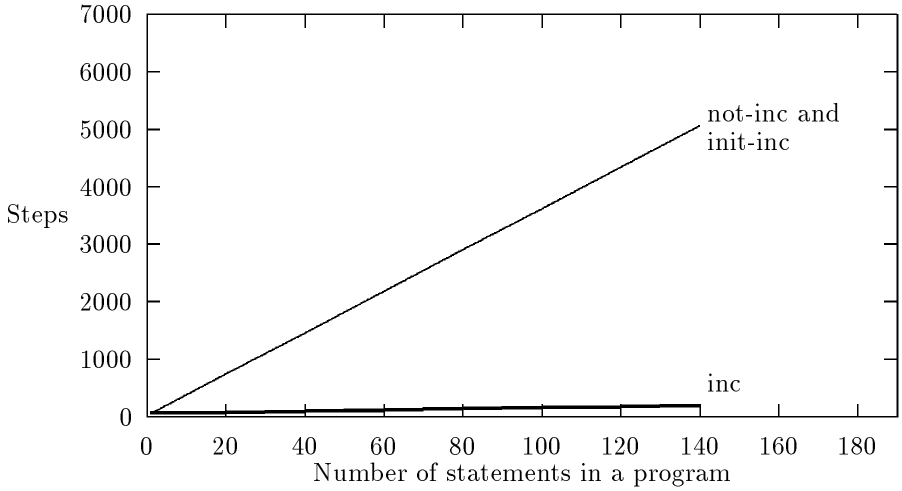


Figure 7.1: Cost of non-incremental type checking of programs (not-inc), incremental type checking without making use of previous results (init-inc), and incremental type checking after a small modification (inc). All programs are type-correct.



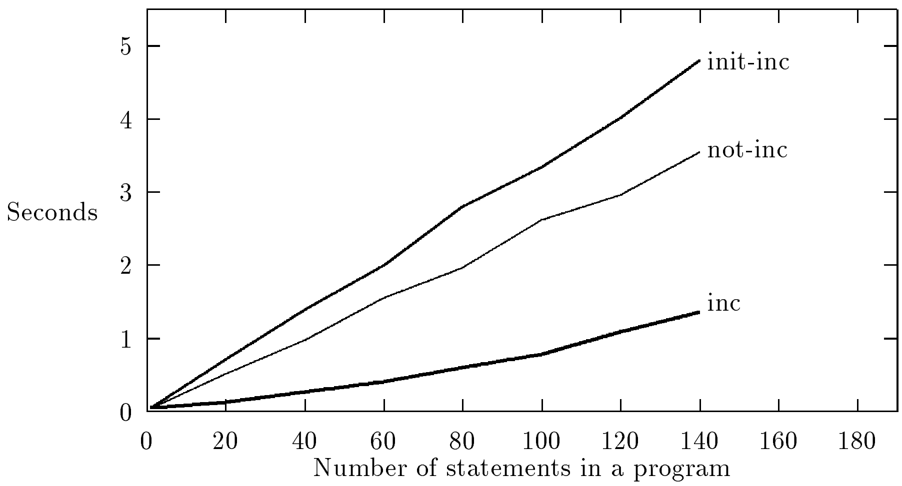


Figure 7.2: Cost of non-incremental type checking of programs (not-inc), incremental type checking without making use of previous results (init-inc), and incremental type checking after a small modification (inc). In each program all statements are incorrect.

The cost for comparing two attribute values is proportional to the size of the smallest value. According to our example specification the type check value of a type correct statement is always true, and so is the type check value of a list of correct statements. Hence the overhead time in Figure 7.1 is very small. The type check value of an incorrect statement in the programs of Figure 7.2 is compatible(natural,string). The type check value of a list of such incorrect statements is compatible(natural,string) & (compatible(natural,string) & ... & (compatible(natural,string))...); its size is proportional to the length of the list. Consequently comparing attribute values is more expensive in this case.

7.2.2 The size of the modification

We want to know the largest modifications for which incremental processing is still profitable. Figure 7.3 and Figure 7.4 present the number of rewrite steps and the time needed for processing modifications of various sizes incrementally.

Description of the test In the test specification the type check functions tcp, tcdecls, tcstms, tcstm, tcexp were declared incremental.

The first test program (Figure 7.3) has 10 declarations and 100 type correct assignments of the form Id1 = Id2. The modifications replace statements at the end of the program by other correct assignments.

The second test program (Figure 7.4) has 10 declarations and 100 assignments Id1 := Id2, in which the type of Id1 is natural and the type of Id2 is string. The modifications replace statements at the end of the program, by incorrect assignments Id1 := Id3, in which the type of Id3 is string.

Discussion Clearly the number of rewrite steps for incremental re-checking a program after a modification of any size is always less than the number of rewrite steps needed for non-incremental type checking.

When considering the time needed we notice that both Figure 7.3 and Figure 7.4 show that in a program with 100 statements the crossover point is at about 70 statements. This means that incremental processing a modification of less than 70 statements is cheaper than completely checking the modified program of 100 statements.

Note that, after completely replacing a program, incremental costs will be equal to the costs for an initial incremental reduction.

7.3 Fine-grain implementation

In this section we pay special attention to modifications in declarations. Whereas the cost of updating after a modification in the statements is linear in the size of the modification, applying the standard, coarse-grain, incremental technique after a modification in the declarations will always cause the whole statements section to be type checked again. Fine-grain incrementality is an extension of the standard incremental technique which is meant to remedy this shortcoming.

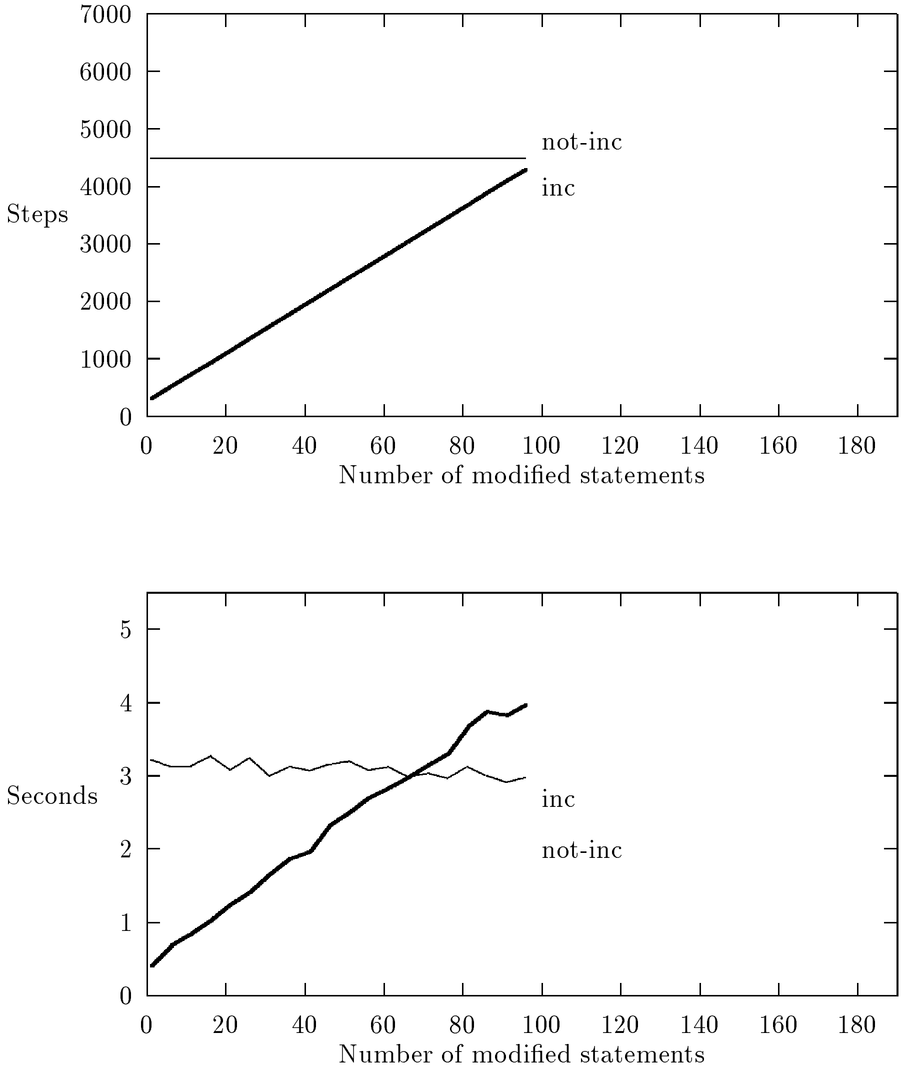


Figure 7.3: Cost of not-incremental and incremental type checking of a program after modifications of various sizes. Before and after the modification the program is type correct.

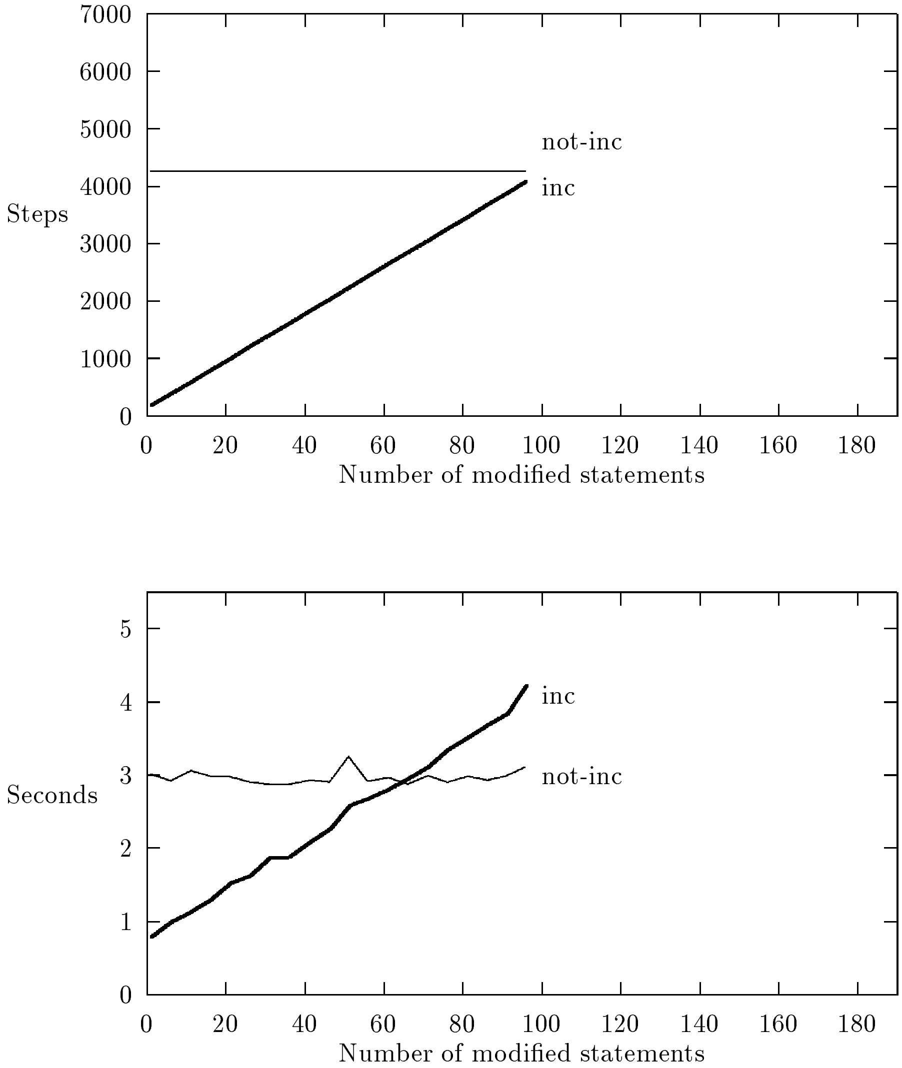


Figure 7.4: Cost of not-incremental and incremental type checking of a program after modifications of various sizes. Before and after the modification all statements are incorrect.

By declaring the lookup function on type environments as incremental (see Appendix C) a modification in the declaration of an identifier causes a re-check only of the statements and expressions in which this identifier actually occurs.

7.3.1 Initial costs

Figure 7.5 shows the initial cost for fine-grain incremental reduction and compares it to the cost for non-incremental reduction and standard incremental reduction.

Description of the test In the test specification the type check functions tcp, tcdecls, tcstms, tcstm, tcexp as well as lookup were declared incremental.

Again we used example programs with 10 declarations and a various number of type correct assignments Id1 = Id2. The use of the 10 declared identifiers is spread evenly over the statements section.

Discussion We already mentioned in Section 7.2.1 that the number of rewrite steps for initial coarse-grain incremental type checking equals the number of rewrite steps for non-incremental type checking. Fine-grain incremental type checking takes less steps due to the fact that the normal form of each lookup Tenv for Id is stored in a function-cache, the first time it occurs. This normal form is used to avoid rewriting of later occurrences of this term.

When considering the time needed we notice that for small programs (less than about 15 assignments) the time for fine-grain reduction exceeds the time for standard incremental reduction. This is caused by the creating of an attribute dependency graph for the lookup function at a Type-environment for each first occurrence of an identifier in the statement section. In larger programs fine-grain reduction is even faster than non-incremental reduction.

7.3.2 Modifying the declarations

Figure 7.6 compares the various strategies after changing the first declaration.

Description of the test We used the same specification and programs as in the previous section. After a program had been checked, we modified the type of its first declaration and re-checked the program.

Discussion After a modification in the declarations standard, coarse-grain, incremental reduction takes almost as many rewrite steps as type checking the complete program, because the complete statements section is to be re-checked. It takes even more time than non-incremental type checking of the complete program, due to the additional overhead for attribute updating.

As expected, the number of rewrite steps is considerably lower for fine-grain incremental rewriting. For programs with more than about 15 statements fine-grain

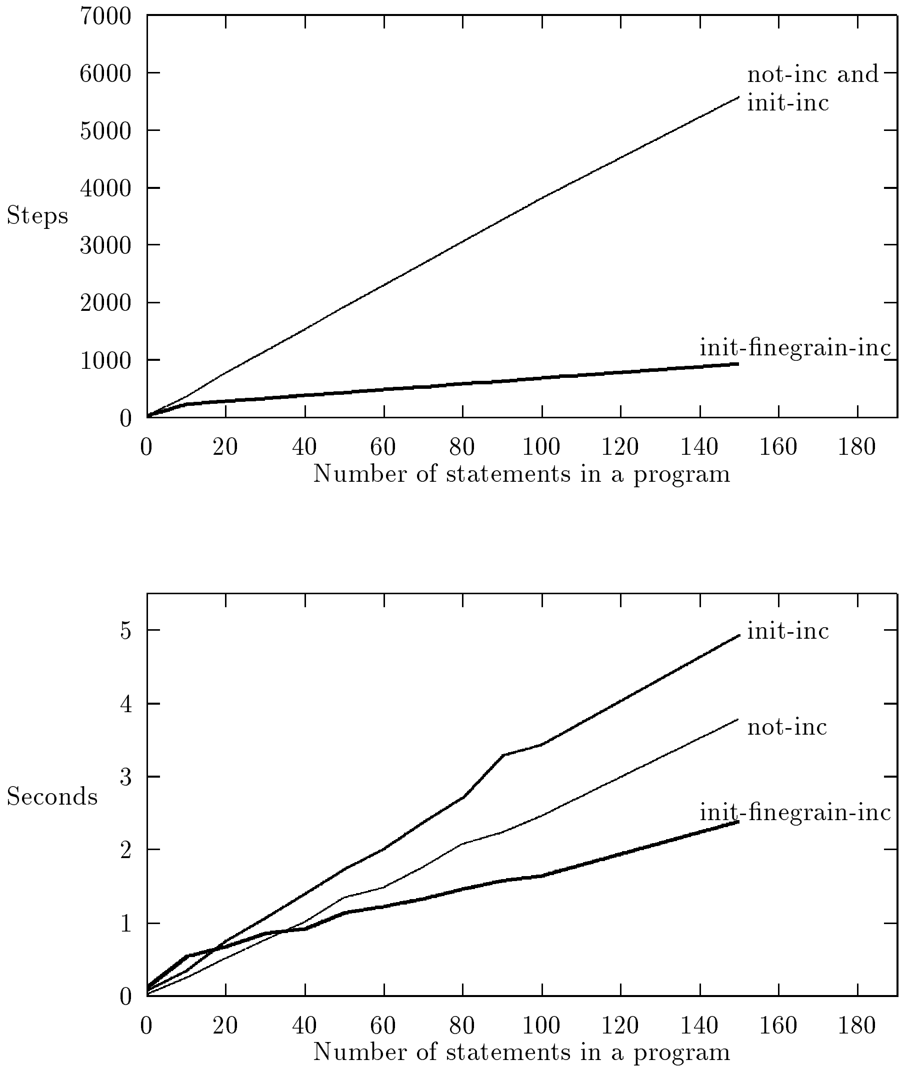


Figure 7.5: Cost of non-incremental (not-inc), coarse-grain incremental (init-inc) and fine-grain incremental (init-finegrain-inc) type checking of programs without using previous results.

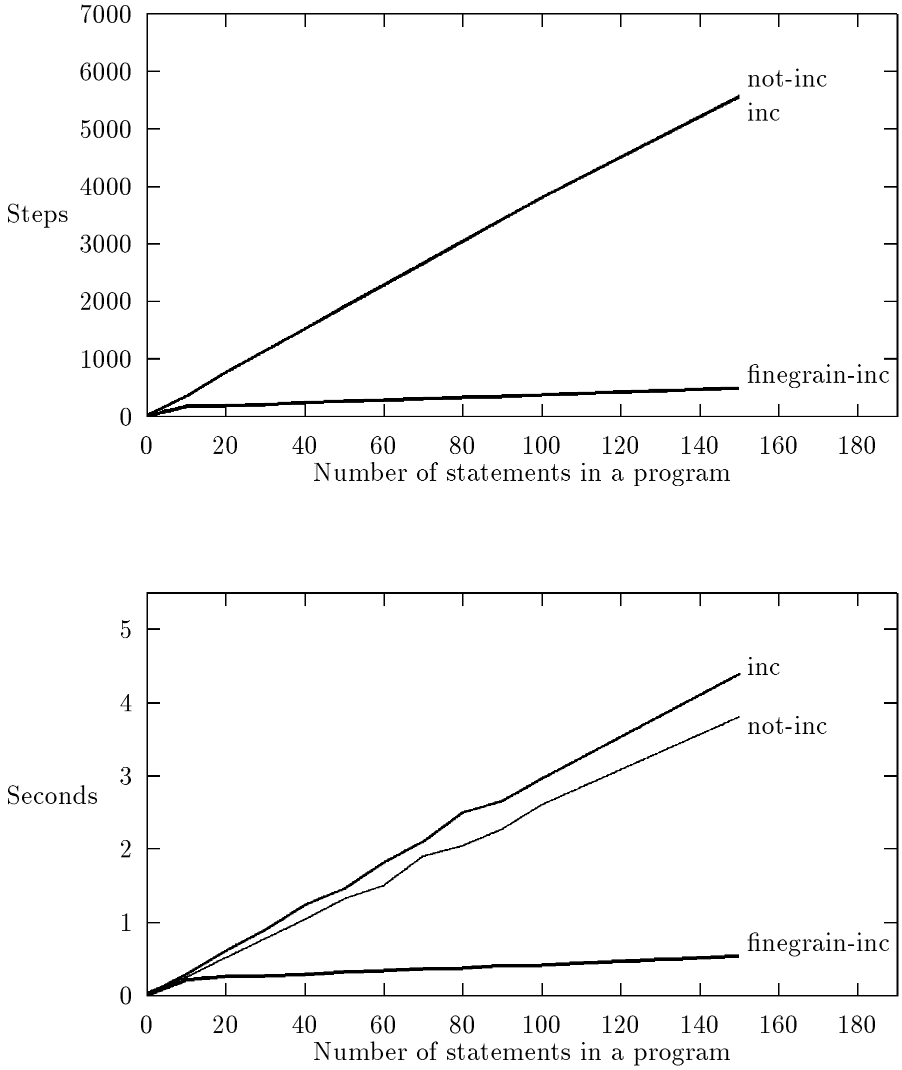


Figure 7.6: Cost of non-incremental (not-inc), coarse-grain incremental (inc) and fine-grain incremental (finegrain-inc) type checking of programs after the type of one identifier has been modified

incremental rewriting is indeed an improvement. The overhead costs that are visible for smaller programs are due to computation of the differences between the old Type-environment and the new one, as well as the subsequent updating of the lookup attributes.

7.4 Lists

ASF+SDF supports lists sorts, and we performed all tests described in this chapter for type check specifications with lists sorts.

The main effect on the results for incremental rewriting has to do with the absence of information on sublists. (See Chapter 4 and the description of the implementation in the previous chapter.)

Because of this, the cost for incremental processing after an element has been modified does not depend on its position, and is always equal to the cost of a modification to the last element of a right-recursive list.

Apart from this, the test results for functions on lists sorts were similar to the ones discussed here.

7.5 Memory

As already mentioned before incremental techniques inevitably use more memory than their non-incremental counterparts, because information has to be stored. Figure 7.7 shows the amount of memory used for incremental rewriting as compared to non-incremental rewriting.

Description of the test In the test specification the type check functions tcp, tcdecls, tcstms, tcstm, tcexp as well as tcidlist and tcid were declared incremental. We used example programs with 10 declarations and an increasing number of correct assignments Id1 = Id2.

The standard garbage collection function in LeLisp [LeL90] provides information on memory-space used. We invoked this garbage collector each time after an editor with a program text was opened, after the program was parsed and after non-incremental or incremental type checking of the program.

We repeated the tests for fine-grain incremental rewriting (by adding the indication incremental for the lookup function) as well as for type incorrect programs.

Discussion The graph for non-incremental reduction indicates the amount of space used for storing the parse tree of the program in the editor. No extra memory is used by non-incremental type checking. The additional amount used for incremental type checking is spent on storing attributes and attribute dependencies.

We did not find any significant difference between the space used for fine-grain or coarse-grain incremental type checking. Nor between the space needed for correct

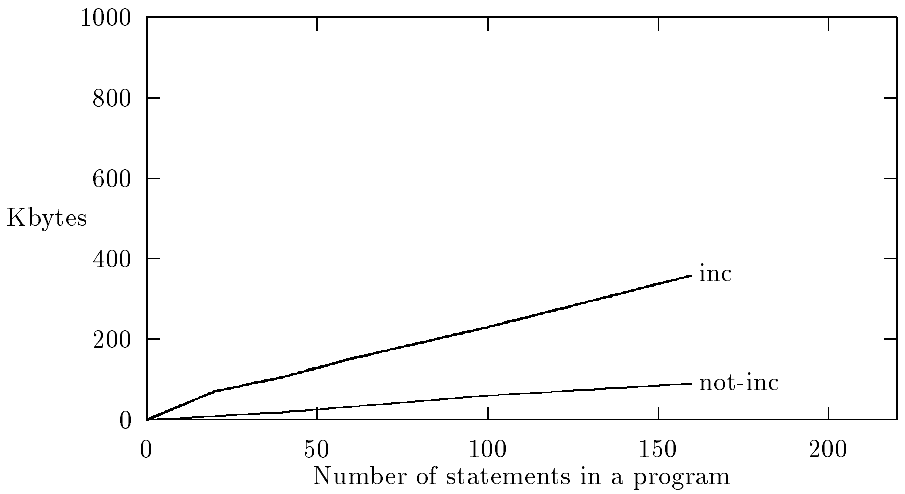


Figure 7.7: Memory used for non-incremental (not-inc), and incremental (inc) type checking of programs

programs (with small attribute values) and incorrect programs (with large attribute values).

7.6 Conclusions

Incremental reduction after a modification is often faster than non-incremental reduction, if the modification is not too large. Fine-grain incrementality is an improvement over standard incremental reduction. Fine-grain incrementality does not only result in efficient incremental behavior after a modification in the declarations, but also causes a *memoization* effect for the `lookup` function.

The overhead for handling attributes is acceptable but could probably be made smaller. The same holds for the amount of memory used.

The tests with incorrect programs made clear that large attribute values reduces the effect of incremental rewriting. It should be investigated whether the use of more detailed information about the dependencies between attribute values and the values of their predecessor attributes can improve this situation. Consider, for instance, an attribute *Att* whose value is a free constructor which has the values of its predecessor attributes as arguments. In this case we could create *constructor dependencies* between subterm of the value of *Att* and the values of its predecessors. When one of the predecessors changes we can make use of these dependencies in updating the attribute value for *Att* by replacing the subterm related to that value. In this way we can avoid the expensive comparison between the old attribute value and the new one.

Chapter 8

Conclusions

8.1 Summary

We have developed a technique for incremental rewriting for a class of algebraic specifications that is especially suited to specifying tools for type checking, program transformation, pretty-printing and compiling programs.

The class is an extension of the class of well-presented primitive recursive schemes with parameters (PRSs). We have exploited the fact that specifications in this class are equivalent to attribute grammars of a certain kind, by storing results of intermediate rewrite steps in attributes of the parse tree of a term being edited. The incremental rewrite strategy is a standard leftmost innermost strategy extended with checks on attributes. If an attribute does not contain a reliable value the normal form of the term being reduced is stored in the attribute. If on the other hand the value in the attribute is reliable this value is used to avoid further reduction of the term.

The technique handles conditional equations in an efficient way, supports specifications with list sorts, and makes use of copy-dependencies between attributes. A simple and elegant extension has been described to allow for efficient processing of modifications in aggregate values like symbol tables.

The technique has been implemented as an extension to the rewrite system of the ASF+SDF meta-environment. Measurements performed for the type checking of a small Pascal like language show that the technique leads to satisfactory incremental behavior.

8.2 Applications

Now that the incremental implementation is part of the ASF+SDF meta-environment, we can give an account of experiences in adapting existing specifications to specifications with incremental functions.

Algebraic specifications offer a lot of liberty to the user. The main adaptation for incremental implementations is to transform the equations for incremental operators so that they meet the restrictions for equations in a layered primitive recursive scheme with conditional equations and list sorts. Error messages and warnings provide suggestions to guide the specification writer.

A very simple modification is to translate equations that are not left-linear into left-linear ones, by adding appropriate conditions.

Equations for incremental functions must be one-step decreasing in their grammar arguments, that is, the first subterm, the grammar argument, of an incremental function in the right-hand side or conditions of an equation must be a direct subterm of the grammar argument in the left-hand side. Often equations do not have this property.

The specification must be well-presentable. Although this notion is difficult to explain it is a very natural one: most PRS-like specifications are well-presentable, Hence, the requirement does not cause many difficulties in practice.

In general, one can say that it is easier to directly write a PRS-like specification with incremental functions than to transform an existing specification into one with incremental functions. In spite of this, specifications have been made incremental with relatively small effort. We mention three examples.

Translation of C to Pim Field has specified in ASF+SDF a translation of a subset of C to Pim. Pim [Fie92] is meant to be used as an intermediate representation for semantic based programming tools.

The structure of the equations for the compile functions were such that after some modifications the functions could be made incremental.

Pretty-printing Van den Brand [Bra93] has specified a box language and transformation rules to generate a default pretty-printer from the ASF+SDF specification of the syntax of any language. The generated pretty-printers consist of *incremental* pretty-print functions that map programs and their substructures onto box terms, as well as defining equations for the pretty-print functions.

Translation of Action Semantics to ASF+SDF Van Deursen and Mosses have specified in ASF+SDF a translation of Action Semantics specifications into ASF+SDF specifications [Mos92, DM93]. The generated environment is a syntax directed editor for Action Semantics, with buttons for translating the Action Semantics specification in the editor into an ASF+SDF specification.

The original specification of the translator was not incremental. It took only half a day to make the changes necessary to make it incremental. For an Action Semantics specification of the dynamic semantics of the toy language Pico, the full translation to ASF+SDF takes 6628 rewrite steps and 10 seconds. After modifications in the equations or import structure of the Action Semantics specification, incremental translation to ASF+SDF is 7-20 times faster.

Besides the incremental translation function, two dump functions were specified in the translator: one for writing the generated ASF+SDF syntax to a file, and another one to do the same for the generated set of ASF+SDF equations. Both dump-syntax and dump-equations first invoke the complete translation function, and then takes the first subterm (the syntax) or the second one (the equations) of the result. An additional advantage of the incremental behaviour of the translation function is that after one `dump` function has been performed the translation result is retained. Consequently, the other `dump` can be performed in one rewrite step.

8.3 Limitations

We list some of the limitations of our approach.

- Our technique for incremental rewriting applies to a restricted class of algebraic specifications only. Algebraic specifications are attractive because of the uniform nature of a specification. It is unfortunate that for incremental rewriting we have to requiring a PRS structure of the specification.

- A restriction related to the previous point is that only a limited control of the granularity of incrementality is offered. Not every arbitrary function can be declared to be incremental.

- The current implementation is based on a leftmost innermost rewriting strategy, which is not always the most efficient one. In order to combine the benefits of outermost strategies with incremental rewriting the algorithms for incremental rewriting would have to be redesigned, but then they would become less efficient.

- The implementation is large and therefore difficult to maintain.

8.4 Further research

It is desirable to adapt the algorithms for incremental rewriting so that the restrictions imposed on specifications for incremental rewriting can be relaxed. There remain possibilities for improving the performance of the specification. Finall, we mention a different approach to achieve incremental rewriting.

Non-local dependencies Often, the writer of a specification is inclined to write equations for incremental functions over an operator with "grand children"; as in equation $\phi(p(q(w_1, w_2)), \ldots) = \psi(w_2, \ldots)$. This would introduce a non-local dependency between attribute for ϕ at the top of p and attributes for ψ at its grand child w_2. Algorithms should be adapted so that incremental rewriting can deal with such non-local dependencies.

Constructor dependencies The performance of the current implementation can be improved by exploiting *constructor dependencies*. Often an attribute value is a free constructor which has the values of its predecessor attributes as arguments. Rather than recomputing this attribute value when a predecessor has changed, one can simply *replace* the subterm related to that value. In this way we can also avoid possible expensive comparisons between the old attribute value and the new one. Constructor dependencies can be thought of as an extension of copy dependencies. Constructor dependencies may be especially effective in two special cases.

(i) The value of an attribute at a *list* node is constructed from the values of attributes in the list elements. In order to compute the constructed value, intermediate steps have to be performed for adding the values one by one. These steps can be avoided when the constructor attribute value is built or updated directly.

(ii) The constructor attribute value is an attributed term itself. When an attributed attribute value changes, the differences between the previous value and the new value, now have to be computed explicitly, by comparing the two values (Chapter 5). When constructor dependencies are used, these differences can be provided immediately.

Incremental rewriting versus memoizing It has been shown that the (fine-grain) incremental implementation of a `lookup` function was not only a successful method for solving the aggregate value problem in incremental reduction, but also resulted in a considerable improvement in the initial reduction, due to a memoizing effect.

It is not a good idea to offer facilities for both incremental functions and memo functions, and let the writer of a specification choose between them. Experience with fine-grain incrementality suggests that a small extension to the code for incremental rewriting will make it possible to combine the two options smoothly. This could be achieved as follows. Assume that a module exists with, for instance, the specification of a table data-type with an incremental `lookup` function. This module can then be used in any specification, and the implementation will make the choice between incremental rewriting or rewriting with memoization:

- If the module is used as an auxiliary module in a specification with incremental functions, which directly call the `lookup` functions, the result is a fine-grain incremental implementation, as described in this thesis.

- If the module is used in an otherwise non-incremental specification or the `lookup`-function is called by non-incremental functions, the implementation will only cause a memo-effect.

Transforming specifications Our implementation of incremental rewriting is very closely connected to one implementation of term rewriting systems. Instead of adapting the term rewriting system like we have done, it may be interesting to *transform specifications* so that any implementation will incrementally rewrite terms over the transformed specification.

An expected disadvantage of this approach is that it will not be as efficient as adapting the underlying implementation.

The advantages of this approach however will be twofold. First, the transformation can be specified in ASF+SDF and it is likely that such a specification is easier to maintain than Lisp code. Secondly, the transformed specification will render rewriting incremental for any implementation. This is especially interesting since Kamperman and Walters [KW93] are currently working on the development of an ASF+SDF-to-C compiler. Compiling a transformed incremental specification will yield a C-program for incremental rewriting.

Appendix A

Correctness of incremental rewriting

We give a correctness proof of the algorithms for incremental rewriting presented in Chapter 2. Our objective is to demonstrate that the result of incremental reduction equals the result of non-incremental reduction.

A.1 Overview of the proof

A tree can be decorated with dependency graphs for several groups of functions, for instance, functions for type checking and function for compilation of a program. In the general case these groups of functions can share certain functions. When attributes in one graph obtain a value during reduction, other attributes of other graphs usually do not. We can therefore not prove that incremental reduction of a term results in a correctly attributed tree. Instead, we introduce the weaker notion of *safe* attributes and a *safely attributed tree* and we will prove that incremental reduction always yields a safely attributed tree.

The proof is constructed as follows. We give a definition of a *safely attributed* tree, and we prove the following steps, for preprocessing of T, reduction of $\phi(T, t_1, \ldots, t_m)$, and updating of T after reduction.

Preprocessing, initial attribution Before any incremental reduction has taken place, T is safely attributed and the status of all attributes is "Initial" (Section A.5.1).

Preprocessing, subtree replacements Subtree replacements in a safely attributed tree T, without attributes with status "Changed", followed by application PROPAGATE-TE-UP as described in Section 2.5.2, yields a safely attributed tree without attributes with status "Changed" (Section A.5.2).

Preprocessing, inherited attributes at the top Resetting the status of inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ of the top node of a safely attributed tree T, followed by applying PROPAGATE-TE-UP as described in Section 2.5.2, yields a safely attributed tree, in which the status of the inherited attributes for ϕ is "Initial" (Section A.5.3).

Reduction Given a safely attributed tree T, in which the status of the inherited attributes for ϕ is "Initial", application of REDUCE-START($\phi(T, t_1, \ldots, t_m), T$) will result in a correct normal form for $\phi(T, t_1, \ldots, t_m)$, and a safely attributed tree T (Section A.7).

Reset-changed Application of RESET-CHANGED to a safely attributed tree T, will result in a safely attributed tree T, without attributes with status "Changed" (Section A.8).

A.2 The dependency graph

We recall some terminology for dependency graphs as has been introduced in Chapter 2, and we list some properties of the dependency graph.

A.2.1 Terminology

Corresponding (inherited) attributes of a synthesized attribute ϕ of a tree T, are the inherited attributes of the same tree that associated with the parameters for the same function ϕ : $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, k)$. The *corresponding* synthesized attribute of an inherited attribute $\mathsf{inh}(\phi, j)$ of a tree T is the attribute ϕ of the same tree. If an inherited attribute is associated with more than one paremeter, see Example 2.9, then its name is a list, e.g. $(\mathsf{inh}(\psi, 1), \mathsf{inh}(\chi, 1))$, and it has several corresponding synthesized attributes: one for each entry in the list.

A direct *upper* successor of an attribute *Att* of a tree T_{Att} is a direct successor of *Att* of the parent tree of T_{Att}. Only synthesized attributes have *upper* successors.

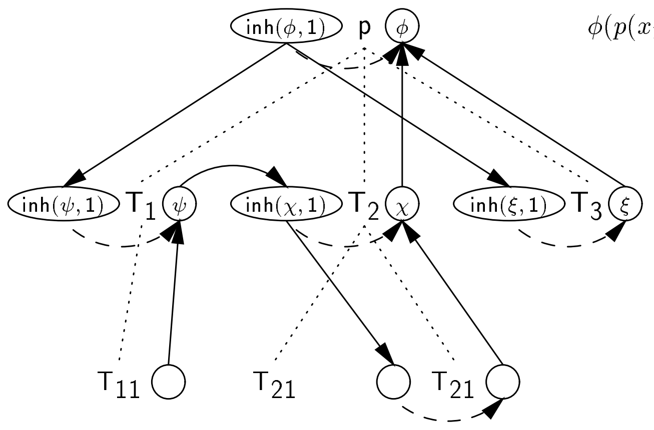
A direct *lower* successor of an attribute *Att* of a tree T_{Att} is a direct successor of *Att* of a subtree of T_{Att}. Only inherited attributes have *lower* successors.

A direct *horizontal* successor of an attribute *Att* of a tree T_{Att} is a direct successor of *Att* either of T_{Att} or of a sibling of T_{Att}. A *horizontal* successor of a synthesized attribute is an inherited attribute. A *horizontal* successor of an inherited attribute is a corresponding synthesized attribute.

Similarly we recognize *upward, downward* and *horizontal* dependencies.

We say that an attribute *Succatt* is on a *upward/horizontal dependency path* from another attribute *Att*, if a dependency path exists from *Att* to *Succatt* that consists only of *upward* and *horizontal* dependencies.

An attribute *Predatt* is a *horizontal/downward predecessor* of an attribute *Att*, if *Att* is on a *upward/horizontal dependency path* from *Predatt*.



$$\phi(p(x_1, x_2, x_3), y) = f(\chi(x_2, \psi(x_1, y)), \xi(x_3, y))$$

Figure A.1: Equation and attribute dependencies

A.2.2 Properties of the dependency graph

The proofs of the following two lemmas follow directly from the construction of the dependency graph as described in Section 2.4.2. We have therefore omitted these proofs. Instead figure A.1 illustrates the lemmas.

The following equation is the defining equation $eq_{\phi,p}$.

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

Lemma A.1
Given
- *T_i is a direct subtree of an attributed tree T with top constructor p,*
- *ϕ is a synthesized attribute of T,*

then ϕ has direct or indirect predecessor attributes ψ and $\mathsf{inh}(\psi, \mathsf{j})$ $(1 \leq j \leq k)$ at a subtree T_i of p, if and only if $\psi(x_i, v_1, \ldots, v_k)$ occurs in the right hand side of the defining equation $eq_{\phi,p}$.

Lemma A.2
Given
- *Att is an inherited or synthesized attribute of a direct subtree T_i of an attributed tree T,*
- *ϕ is a synthesized attribute at T, and a direct or indirect successor of Att,*

then the set of all predecessors of Att that are attributes of T, is a subset of all corresponding inherited attributes of ϕ: $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, k)$.

When proving properties of an attribute w.r.t. a indirect synthesized predecessor in a possibly remote part of the tree, we will often make use of *intermediate* attributes, that is, attributes that are on a dependency path from that predecessor to the attribute under consideration.

Lemma A.3
Given

- *Att is an attribute of* T_{Att}*, a nested subtree of an attributed tree* T*,*
- *Att is an indirect successor of a synthesized attribute Synatt in the context of* T_{Att} *in* T*,*
- *Att is* not *on a upward/horizontal dependency path from Synatt,*

then an inherited attribute Inhatt exists, which is on a upward/horizontal dependency path from Synatt, and a predecessor of Att. Moreover, Inhatt is an attribute of an ancestor of T_{Att}*.*

Proof Since *Att* is a successor of *Synatt* that is *not* on a upward/horizontal dependency path from *Synatt*, at least one *downward* dependency occurs in every path from *Synatt* to *Att*. Let *Inhatt* be the origin of the first *downward* dependency that appears when following the shortest path from *Synatt* to *Att*. *Inhatt* is a predecessor of *Att* and it lies on a upward/horizontal dependency path from *Synatt*.

Let T_{Inh} be the tree of *Inhatt*. Assume that *Att* is *not* an attribute of a subtree of T_{Inh}. Then *Att* is either an attribute of T_{Inh} itself or an attribute in the context in T of T_{Inh}. In either case the corresponding synthesized attribute of *Inhatt* is on the path from *Inhatt* to *Att*. This leads to a contradiction, because in that case a shorter path exist from *Synatt* to *Att* which follows the *horizontal* dependency from *Inhatt* to this corresponding attribute.

Therefor, *Att* must be attribute of a subtree of T_{Inh}. Hence, *Inhatt* is an attribute of an ancestor of T_{Att}. □

A.3 Correct attributes and correctly attributed trees

In a PRS, the value of a synthesized attribute ϕ of a tree with top constructor p is determined by the right-hand side of the defining equation $eq\phi, p$

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

Let $\psi(x_i, v_1, \ldots, v_k)$ be a subterm of τ. The value of an inherited attribute $\mathsf{inh}(\psi, \mathsf{j})$ of a subtree tree T_i of constructor p is determined by the parameter term v_j.

We give definitions of correct attributes values and correctly attributed trees. Note that these definitions only concern the *value* of an attribute and not their status. The *status* of attributes plays a role in the notion of *safe attributes* which will be defined in the next section.

Definition A.4 *An attribute Att of a subtree* T_{Att} *of a tree* T *is correct w.r.t.* T *and the inherited attributes of* T*, if a function* ϕ *for* T *exists, such that during reduction of* $\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots, value_{\mathsf{inh}}(\phi, m))$ *the normal for of the term associated with Att equals* $value_{Att}$*.* □

Definition A.5 *A tree T is correctly attributed if all its attributes are correct w.r.t. the tree and its inherited attributes.* □

We will often use the phrase "correct w.r.t. T" as an abbreviation for "correct w.r.t. T and the inherited attributes of T".

An attribute can be "correct w.r.t. its direct predecessors". Note that the direct predecessors of an attribute are derived from one defining equation.

Definition A.6 *Let τ be the right-hand side of the defining equation $eq_{\phi,p}$. A synthesized attribute ϕ of a tree T with constructor p is correct w.r.t. its direct predecessors, if $value_\phi$ equals the normal form obtained by replacing in τ the associated terms of these predecessors by the attribute values.* □

Definition A.7 *Let $\psi(x_i, v_1, \ldots, v_k)$ be a subterm of the right-hand side τ of the defining equation $eq_{\phi,p}$. An inherited attribute $\mathsf{inh}(\psi, j)$ of a subtree T_i of constructor p is correct w.r.t. its direct predecessors, if $value_{\mathsf{inh}}(\psi, i)$ equals the normal form obtained by replacing in v_j the associated terms of these predecessors by the attribute values.* □

An attribute can be "partially correct w.r.t. one direct predecessor".

Definition A.8 *Let τ be the right-hand side of the defining equation $eq_{\phi,p}$. A synthesized attribute ϕ of a tree T with constructor p is correct w.r.t. one direct predecessors, if after replacing in τ the associated term of this predecessor by the attribute value of the predecessor, values for the remaining variables in τ can be found so that the normal form of τ equals $value_\phi$.* □

Definition A.9 *Let $\psi(x_i, v_1, \ldots, v_k)$ be a subterm of the right-hand side τ of the defining equation $eq_{\phi,p}$. An inherited attribute $\mathsf{inh}(\psi, j)$ of a subtree T_i of constructor p is correct w.r.t. one direct predecessor, if after replacing in v_j the associated term of this predecessor by the attribute value of the predecessor, values for the remaining variables in v_j can be found so that the normal form of v_j equals $value_{\mathsf{inh}}(\psi, j)$.* □

We will often make use of the fact that if an attribute is correct w.r.t. a tree, it is also correct w.r.t. a subtree of that tree, provided inherited attributes at the top of that subtree are correct in the context tree. Under the same conditions it holds that if an attribute is correct w.r.t. a subtree, it is also correct in the context tree of that subtree.

Lemma A.10
Given
- *T_i is a direct subtree of an attributed tree T,*
- *Att is an attribute at a (nested) subtree T_{Att} of T_i,*
- *Att is correct w.r.t. T and the inherited attributes of T,*
- *all inherited predecessors of Att at the top of T_i are correct w.r.t. T,*

then Att is correct w.r.t. T_i and the inherited attributes of T_i.

Proof According to Definition A.4 of correct attributes, a function ϕ on T exists, such that during reduction of $\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots, value_{\mathsf{inh}}(\phi, m))$, the normal form of the associated term of *Att* equals the value of *Att*.

Let ψ be an attribute of T_i s.t. ψ is a direct or indirect successor of *Att* and a predecessor of ϕ.

Reduction of $\psi(T_i, value_{\mathsf{inh}}(\psi, 1), \ldots, value_{\mathsf{inh}}(\psi, k))$ is a substep of the reduction of $\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots, value_{\mathsf{inh}}(\phi, m))$.

Since *Att* is a predecessor of ψ, the associated term of *Att* will appear in the reduction of $\psi(T_i, value_{\mathsf{inh}}(\psi, 1), \ldots, value_{\mathsf{inh}}(\psi, k))$. The normal form of this term equals the value of *Att*. Hence, *Att* is correct w.r.t. T_i and the inherited attributes of T_i. □

Lemma A.11
Given

- *T_i is a direct subtree of an attributed tree T,*
- *Att is an attribute at a (nested) subtree T_{Att} of T_i,*
- *Att is correct w.r.t. T_i and the inherited attributes of T_i,*
- *all inherited attributes of Att at the top of T_i are correct w.r.t. T and the inherited attributes of T,*

then Att is correct w.r.t. T and the inherited attributes of T.

Proof An attribute ϕ of T exists that is a successor of ψ. The attributes $\mathsf{inh}(\psi, 1)$, $\ldots, \mathsf{inh}(\psi, k)$ are correct w.r.t. T. So, during reduction of $\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots,$ $value_{\mathsf{inh}}(\phi, m))$ the term $\psi(T_i, s_1, \ldots, s_k)$ will appear. And the normal form of each s_i equals $value_{\mathsf{inh}}(\phi, i)$.

Since we use innermost reduction, reduction will then procceed with the term $\psi(T_i,\ value_{\mathsf{inh}}(\psi, 1), \ldots, value_{\mathsf{inh}}(\psi, k))$. The associated term of *Att* will appear in this reduction and the normal form of this term equals the value of *Att*. □

A.4 Safe attributes and safely attributed trees

Incremental reduction of a term does not always result in a correctly attributed tree. Therefore we introduce the weaker notions of *safe* attributes and a *safely attributed tree*. Intuitively speaking, an attribute is safe if it either has a correct value, or its own status or the status of its predecessors indicate that its value must be recomputed.

Definition A.12 *Let T be an attributed tree. An attribute Att in T is* safe *if it has one of the following properties:*

safe-a *Its status is "Initial".*

safe-b *Its status is "TobeEvaluated", and its value is correct w.r.t its direct predecessors with status "Initial", "TobeEvaluated", "Unchanged".*

safe-c *Its status is "Unchanged" or "Changed", its value is correct w.r.t. its direct predecessors, and its value is correct w.r.t. T and its inherited attributes.*

safe-d *Its status is "Unchanged" or "Changed" and its value is correct w.r.t. its direct predecessors, and it has an (indirect) inherited predecessor with status "Initial" or "TobeEvaluated" at an ancestor node of T_{Att}.*

□

Definition A.13 *An attributed tree T is* safely attributed *if each attribute is safe in T, and moreover, all successor attributes on a upward/horizontal dependency path from an attribute with status "Initial" or "TobeEvaluated" have status "Initial" or "TobeEvaluated".* □

Note that on the one hand, in a safely attributed tree all upward and horizontal successors on a upward/horizontal dependency path from a "TobeEvaluated" or "Initial" attribute have status "TobeEvaluated" or "Initial", whereas on the other hand, the status of the remaining successors can be "Unchanged" or "Changed".

Lemma A.14 *In a safely attributed tree, all lower/horizontal predecessors of an attribute with status "Unchanged" or "Changed", have status "Unchanged" or "Changed".*

Proof Follows directly from the property that all upward/horizontal successors of an attribute with status "Initial" or "TobeEvaluated" have status "Initial" or "TobeEvaluated". □

Lemma A.15
Given
- *T_{Att} is a direct or nested subtree of a safely attributed tree T,*
- *Att is an attribute of T_{Att},*
- *Att has property safe-c,*

then all inherited predecessors of Att at ancestors of T_{Att} have property safe-c.

Proof Let *Inhatt* be an inherited predecessor of Att at an ancestor T_{Inh} of T_{Att}. Since Att has property *safe-c*, the status of *Predatt* is "Unchanged" or "Changed", so it either has property *safe-c* or *safe-d*. We show that *Inhatt* cannot have property *safe-d*.

If *Inhatt* would have property *safe-d*, then there is an inherited predecessor of *Inhatt* with status "TobeEvaluated" or "Initial" at an ancestor tree of T_{Inh}. But then this ancestor node of T_{Inh} is also an ancestor of T_{Att}, and the inherited predecessor of *Inhatt* is also a predecessor of Att so Att would have had property *safe-d*. Hence, *Inhatt* has property *safe-c*. □

We will often make use of the fact that if an attribute is safe in a tree T, it is also safe w.r.t. the subtrees it resides in. And, vice versa, if an attribute is safe w.r.t. a subtree of an attributed tree, then under certain restrictions it is also safe in the context tree. These properties are shown in the following two lemmas.

Lemma A.16
Given
- *T_i is a direct subtree of a safely attributed tree T,*
- *Att is an attribute at a (nested) subtree T_{Att} of T_i,*
- *Att is safe in T,*

then Att is safe in T_i.

Proof
(1) Attributes in T_i, that have no predecessors in T are safe in T_i.

(2) For attributes in T_i that do have predecessors in T we distinguish four cases according to the safety property of Att in T.

(2.(*safe-a*)) Att has status "Initial" so it has property *safe-a* in T_i as well.

(2.(*safe-b*)) Att has status "TobeEvaluated" and it is correct w.r.t. its direct predecessors with status "TobeEvaluated", "Initial" and "Unchanged". The direct predecessor s of Att in T_i are a subset of the direct predecessors of Att in T. So Att is *safe-b* in T_i.

(2.(*safe-c*)) Att has status "Unchanged" or "Changed" and it is correct w.r.t. its direct predecessors. In particular it is correct w.r.t. its direct predecessors in T_i.

According to Lemma A.15, all inherited predecessors of Att at the top of T_i have property *safe-c* in T. So all inherited attributes at the top of T_i are correct w.r.t T. According to Lemma A.10 Att is correct in T_i.

Hence, Att has property *safe-c* in T_i.

(2.(*safe-d*)) Att has status "Unchanged" or "Changed". Att is correct w.r.t. its direct predecessors in T. In particular it is correct w.r.t. its direct predecessors in T_i. Att has an inherited predecessor attribute in T with status "TobeEvaluated" or "Initial".

(2.(*safe-d*).1) If Att has an inherited predecessor attribute in T_i with status "TobeEvaluated" or "Initial", it has property *safe-d* in T_i.

(2.(*safe-d*).2) If Att has no inherited predecessor attribute in T_i with status "TobeEvaluated" or "Initial", then all its inherited predecessors at the top of T_i have status "Unchanged" or "Changed", since they are correct w.r.t. T_i they have property *safe-c* in T_i.

We proceed by induction over the length of the longest dependency path between Att and its inherited predecessors at the top of T_i.

(2.(*safe-d*).2.1) Assume the distance is 1. Att is a direct predecessor of inherited attributes at the top of T_i. Consequently, Att is an inherited attribute at a deirect subtree of T_i.

Other predecessors of Att in T_i are attributes without any predecessor at the top of T_i. Therefore they are safe in T_i. Moreover they are downward or horizontal predecessors. According to Lemma A.14, they all have status

"Unchanged" or "Changed". So, they have property *safe-c* in T. and they are correct w.r.t. T. According to Lemma A.10, they are also correct in T_i.

Att is correct w.r.t. T_i, because it is correct w.r.t. all its direct predecessors and all these predecessors are correct in T_i Hence *Att* has property *safe-c*.

(2.(*safe-d*).2.2) Assume the distance is N and it has been proved that attributes with distances $< N$ have property *safe-c*.

All direct predecessors of *Att* either have distance $< N$ or they have no predecessor at the top of T_i. In the first case they are correct in T_i by induction assumption. In the second case they are correct in T_i according to Lemma A.10.

Att is correct w.r.t. T_i, because it is correct w.r.t. all its direct predecessors and all these predecessors are correct in T_i. Hence *Att* has property *safe-c*.

□

Lemma A.17
Given
- *T_i is a direct subtree of a safely attributed tree T,*
- *Att is an attribute at a (nested) subtree T_{Att} of T_i,*
- *Att is safe w.r.t. T_i,*
- *all inherited predecessors of Att at the top of T_i, are safe in T,*

then Att is safe in T.

Proof

(1) If *Att* has no predecessor at the top of T_i, *Att* is clearly safe in T.

(2) If *Att* has a predecessor at the top of T_i, we distinguish four cases according to the safety property of *Att* in T_i.

(2.(*safe-a*)) *Att* has status "Initial" so it has property *safe-a* in T as well.

(2.(*safe-b*)) *Att* has status "TobeEvaluated" and it is correct w.r.t. its direct predecessors with status "TobeEvaluated", "Initial" and "Unchanged". *Att* has no direct predecessors in the context of T_i in T, so *Att* is *safe-b* in T.

(2.(*safe-c*)) *Att* has status "Unchanged" or "Changed" and it is correct w.r.t. its direct predecessors. *Att* has no direct predecessors in the context of T_i in T, so these two aspects remain valid in T. *Att* is correct w.r.t. T_i and its inherited predecessors attributes at the top of T_i. These inherited predecessors have property *safe-c* in T_i (Lemma A.15). Hence they have status "Unchanged" or "Changed".

(2.(*safe-c*).1) If all these inherited attributes are correct in T, then *Att* is also correct w.r.t. T (Lemma A.11). Hence it has property *safe-c* in T.

(2.(*safe-c*).2) If one of these inherited attributes is not correct in T, then it has property *safe-d*. So, it has an inherited predecessor in T with status "Tobe-Evaluated" or "Initial". This inherited predecessor is also a predecessor of *Att*. Hence, *Att* has property *safe-d* in T.

(2.(*safe-d*)) *Att* has status "Unchanged" or "Changed". It is correct w.r.t. its direct predecessors. There is an inherited predecessor attribute with status "Tobe-Evaluated" or "Initial". *Att* has no direct predecessors in the context of T_i in T, and all predecessors of *Att* in T_i are also predecessors of *Att* in T. So all aspects remain valid in T. Hence *Att* is *safe-d* in T.
□

A.5 Preprocessing

In this section we show that before incremental reduction of a term $\phi(T,\, t_1, \ldots, t_m)$ starts, T is safely attributed. Preprocessing is described in Section 2.5.2.

If the term $\phi(T, t_1, \ldots, t_m)$ is to be reduced before any incremental reduction has taken place, the tree T is decorated with attributes with status "Initial". We call this a tree with an initial attribution.

Otherwise, we assume that a safely attributed tree T' exists, and that subtrees are replaced in T' to obtain T. The new attributes in a new subtree, as well as the synthesized attributes at the place of subtree replacement all obtain status "Initial". The algorithm PROPAGATE-TE-UP of Figure 2.11 is applied to the synthesized attributes at the top of the new subtrees.

Next, the inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ at the top of T obtain status "Initial" and PROPAGATE-TE-UP is applied to these attributes as well.

A.5.1 Initial attributed tree

Theorem A.18 *A tree T with an initial attribution is safely attributed and has no attributes with status "Changed".*

Proof All attributes have status "Initial" and are therefore safe. The successors of an attribute with status "Initial" all have status "Initial" as well. □

A.5.2 Subtree replacements

Theorem A.19 *Let T' be safely attributed tree, and assume that no attributes in T' have status "Changed". When subtrees are replaced in T' yielding T, and the algorithm* PROPAGATE-TE-UP *is applied to the synthesized attributes at the top of the new subtrees, then*

-a- *the resulting tree T is safely attributed;* **-b-** *no attributes in T have status "Changed".*

Proof It suffices to prove the theorem for one subtree replacement. Let *Newsub* be the new subtree.

(1) We first prove that all attributes in T are safe.

(1.1) All new attributes of subtrees of *Newsub*, as well as the synthesized attributes at the top of *Newsub* are safe because they have status “Initial”.

(1.2) All attribute that have no predecessor in *Newsub* are safe, because they were safe in T'.

(1.3) The remaining attributes are attributes in the context tree of *Newsub* that are successors of attributes in *Newsub*. In particular, they are successors of synthesized attributes of the top node of *Newsub*. Let *Att* of tree T_{Att} be a successor of the synthesized attribute *Synatt* of *Newsub*.

(1.3.1) Assume *Att* is on a upward/horizontal dependency path from *Synatt*. If it is visited by PROPAGATE-TE-UP then its new status is “TobeEvaluated”. The algorithm PROPAGATE-TE-UP does not change the status of attributes with status “TobeEvaluated” or “Initial”, nor does it proceed with their successors. However, it follows from the fact that T' was safely attributed, that all these successors already have status “TobeEvaluated” or “Initial”. So *Att* either has status “Initial”, or it has status “TobeEvaluated”. We distinguish four cases, according to the previous safety property of *Att*:

(1.3.1.(*safe-a*)) Its previous status was “Initial”, then it is still “Initial”, so *Att* still has property *safe-a*

(1.3.1.(*safe-b*,*safe-c*,*safe-d*)) In each of these cases the value of *Att* was correct w.r.t. to all direct predecessors, (since there are no “Changed” predecessors). The value of none of these predecessors has changed, so this is still the case. The new status is “TobeEvaluated”, hence *Att* has property *safe-b*.

(1.3.2) Assume *Att* is *not* on a upward/horizontal dependency path from *Synatt*. We distinguish four cases according to the previous safety property of *Att*.

(1.3.2.(*safe-a*)) Its previous status was “Initial”, then it is still “Initial”, so *Att* still has property *safe-a*

(1.3.2.(*safe-b*)) The status of *Att* remains “TobeEvaluated”. The value of *Att* was correct w.r.t. to all direct predecessors, The value of none of these predecessors has changed, so this is still the case. Hence *Att* has property *safe-b*.

(1.3.2.(*safe-c*),(*safe-d*)) The status of *Att* remains “Unchanged”. The value of *Att* was correct w.r.t. to all direct predecessors, The value of none of these predecessors has changed, so this is still the case.

Lemma A.3 states that an inherited attribute *Inhatt* exists that is a predecessor of *Att* and that resides on a upward/horizontal dependency path from *Synatt*. The status of *Inhatt* is “TobeEvaluated” or “Initial”, therefore *Att* has property *safe-d*.

(2) Secondly, we have to prove that all attributes on a upward/horizontal dependency path of an “Initial” attribute have status “TobeEvaluated” or “Initial”.

(2.1) Attributes of subtrees of *Newsub* all have status “Initial”. Their upward/ horizontal successors are either attributes within *Newsub*, in which case their status is “Initial”, or upward/horizontal successors of synthesized attributes at the top of *Newsub*. We already demonstrated that the status of these latter successors is “Tobe-Evaluated” or “Initial”.

(2.2) The status of upward/horizontal successors of “Initial” and “TobeEvaluated” attributes in the context of *Newsub*, was "TobeEvaluated" or "Initial" in T'. Subtree replacement nor PROPAGATE-TE-UP has changed this.

(3) Finally, no attributes in T have status "Changed", since no attributes in T' had this status, and the new attributes have status "Initial". PROPAGATE-TE-UP clearly has not altered the status of any attribute into "Changed".
□

A.5.3 Inherited attributes at the top of T

Theorem A.20 *Let T be safely attributed, and assume that no attributes have status "Changed". When the status of inherited attributes* $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ *at the top node of T obtain status "Initial" and* PROPAGATE-TE-UP *is applied to these attributes, then*

-a- *the resulting tree T is safely attributed;*

-b- *the status of all* $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ *is "Initial".*

Proof

(1) First, we prove that all attributes in T are safe. PROPAGATE-TE-UP only visits their corresponding synthesized attributes. If the status of such a synthesized attribute, *Synatt*, was "Initial" or "TobeEvaluated" it remains so. Otherwise it is reset to "TobeEvaluated".

(1.1) The inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ at the top of T are safe because they have status "Initial".

(1.2) Every successor *Succatt* of an inherited attribute $\mathsf{inh}(\phi, \mathsf{j})$ is safe. We distinguish four cases according to the previous safety property.

(1.2.(*safe-a*)) Its status was "Initial" and remains "Initial". Hence, *Succatt* still has property *safe-a*

(1.2.(*safe-b*)) Its status remains "TobeEvaluated" and its value remains correct with respect to its "TobeEvaluated", "Initial" and "Unchanged" predecessors, since no attribute value has been changed. Hence *Succatt* still has property *safe-b*.

(1.2.(*safe-c*)) Its value remains correct with respect to all its predecessors. Its status was "Unchanged".

If its new status is "TobeEvaluated", *Succatt* has property *safe-b*.

Otherwise its status is still "Unchanged". Since neither T nor any inherited attribute value at the top of T has changed, *Succatt* is still correct w.r.t. T. Hence, *Succatt* has property *safe-c*.

(1.2.(*safe-d*)) Note that an attribute at a top node never has property *safe-d*. So, *Succatt* is an attribute in a subtree of T. Its status remains "Unchanged". Its value remains correct with respect to all its predecessors. An inherited predecessor attribute at an ancestor node had status "TobeEvaluated" or "Initial". The status of no such attribute has been altered. Hence *Succatt* still has property *safe-d*.

(1.3) Attributes that are not successors of one of the attributes $\mathsf{inh}(\phi, 1), \ldots,$ $\mathsf{inh}(\phi, \mathsf{m})$ remain safe.

(2) Secondly, we have to prove that all attributes on a upward/horizontal dependency path from attributes with status "Initial" have status "TobeEvaluated" or "Initial". For the inherited attributes at the top with status "Initial" this is the result of PROPAGATE-TE-UP. For all other "Initial" attributes this was the case before, and the status of none of their "TobeEvaluated" or "Initial" successors has been altered.

(3) Finally, it is clear that all inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, \mathsf{m})$ have status "Initial".
□

A.6 Store and propagate

In proving properties of the reduction algorithms, we will make use of the results proved in this section. Namely that applying the algorithms STORE-INH-ATT and STORE-SYN-ATT of Figure 2.10 for storing a correct attribute value in an attribute of a safely attributed tree, and propagating status information preserves safety of the tree. The algorithm PROPAGATE-UNCHANGED of Figure 2.13 is invoked by both STORE algorithms. We first prove that application of PROPAGATE-UNCHANGED preserves safety of an attributed tree.

A.6.1 Propagate Unchanged

Lemma A.21 *Let Att-uc be an attribute with status "Unchanged" in a safely attributed tree T, then after application of* PROPAGATE-UNCHANGED*(Att-uc), T is safely attributed.*

Proof
(1) No attribute changes when *Att-uc* has no successor with status "TobeEvaluated" or when all "TobeEvaluated" successors of *Att-uc* have at least one predecessor of which the status is not "Unchanged". When nothing changes T remains safely attributed.

(2) Assume, that *Att-uc* has successors, whose status is reset to "Unchanged", and PROPAGATE-UNCHANGED is recursively applied to these attributes. It suffices to prove, that after the status of one of these successors, say *Att* of tree T_{Att}, has been set to "Unchanged", T is safely attributed.

Att's previous status was "TobeEvaluated". All its predecessors have status "Unchanged".

(2.1) *Att* itself is safe and has either property *safe-c* or *safe-d*: *Att*'s previous status was "TobeEvaluated", so it had property *safe-b*, hence its value was correct w.r.t. all its "Unchanged" predecessor.

(2.1.1) If all these predecessors are correct, the value of *Att* is correct as well, and *Att* has property *safe-c*.

(2.1.2) Otherwise, at least one predecessor is not correct:

(2.1.2.1) Assume that one of its direct upper predecessors, *Inhatt*, has an incorrect value. *Inhatt* is safe and has status "Unchanged", so it has property *safe-d*: there is a predecessor with status "Initial" or "TobeEvaluated", at an ancestor node of the tree for *Inhatt*. This predecessor is also a predecessor of *Att*. Hence, *Att* has property *safe-d*.

(2.1.2.2) Assume that one of its direct lower or horizontal predecessors, *Predatt*, is not correct. *Predatt* is either (a) a synthesized attribute at a sibling of T_{Att}, or (b) an inherited attribute at T_{Att}, or (c) a synthesized attribute at a subtree of T_{Att}. *Predatt* has status "Unchanged" and its value is incorrect, so *Predatt* has property *safe-d*. Which, in turn, means that there is an inherited predecessor *Inhatt* of *Predatt* with status "Initial" or "TobeEvaluated", at an ancestor node of the tree for *Predatt*. This predecessor is also a predecessor of *Att*. We distinguish three cases according to the position (a), (b) or (c) of *Predatt*:

(2.1.2.2.(a),(b)) *Predatt* is an attribute at a sibling of T_{Att} or at T_{Att} itself. The inherited predecessor *Inhatt* is therefore a predecessor attribute at an ancestor node of T_{Att}, rendering *Att* to have property *safe-d* as well.

(2.1.2.2.(c)) *Predatt* is a synthesized predecessor attribute at a subtree of T_{Att}, hence *Att* is a synthesized attributee as well. According to Lemma A.2, the inherited predecessors of *Predatt* at T_{Att} are the corresponding inherited attributes of *Att*.

Because inherited attributes of *Att* are direct predecessors of *Att*, they have status "Unchanged" by assumption. So, if *Predatt* has property *safe-d*, *Inhatt* is an inherited attribute at an ancestor of T_{Att}. In that case *Att* has property *safe-d* as well.

(2.2) All direct and indirect successors of *Att* remain safe. We classify them according to their previous safety property. Note that only the status of *Att* has changed and no attribute value has changed.

(2.2.(*safe-a*)) Attributes with status "Initial" still have this status. Hence they keep property *safe-a*.

(2.2.(*safe-b*)) The status of "TobeEvaluated" attributes remains "TobeEvaluated". No attribute value has been changed, so they remain correct w.r.t. their predecessors. Hence they keep property *safe-b*.

(2.2.(*safe-c*)) The status of *Att* remains "Unchanged" or "Changed". Its value has not changed, not has the value of its predecessors. So it remains correct w.r.t. its predecessors, and w.r.t. the tree. Hence they keep property *safe-c*.

(2.2.(*safe-d*)) We distinguish attributes *outside* T_{Att} and attributes *inside* T_{Att}.

(2.2.(*safe-d*).1) For an attribute *Succatt outside* T_{Att} at some tree T_{Succ} an inherited predecessor with status "Initial" or "TobeEvaluated" exists at a context tree of T_{Succ}. This inherited attribute is not *Att*. So its status remains "Initial" or "TobeEvaluated". Hence *Succatt* keeps property *safe-d*.

(2.2.(*safe-d*).2) An attribute *Succatt inside* T_{Att} at some tree T_{Succ} has status "Unchanged" or "Changed", and its value is correct with respect to their direct predecessors.

(2.2.(*safe-d*).2.1) Assume *Succatt* still has an inherited predecessor at an ancestor node with status "TobeEvaluated" or "Initial", *Succatt* has property *safe-d*.

(2.2.(*safe-d*).2.2) Assume *Succatt* has no inherited predecessor at an ancestor node with status "TobeEvaluated" or "Initial". We prove that *Succatt* now has property *safe-c*: We only need to prove that the value of its direct predecessors is correct.

(2.2.(*safe-d*).2.2.1) All (direct) lower and horizontal predecessors of *Succatt* have status "Unchanged" or "Changed" (Lemma A.14). None of these attributes has property *safe-d*, because if they would then *Succatt* would have property *safe-d* as well, which contradicts our assumption. So all lower and horizontal predecessors have property *safe-c*. Hence they have a correct value.

(2.2.(*safe-d*).2.2.2) All upper direct predecessors of *Succatt* are inherited attributes, of the parent node of the tree of *Succatt*. Since we assume that *Succatt* does not have property *safe-d*, the status of these inherited attributes is "Unchanged" or "Changed". (Lemma A.15). Moreover, if these inherited attributes would have property *safe-d* so would *Succatt*, hence they have property *safe-c*. Hence their value is correct.

(2.3) Attributes that are not successors of *Att* are not affected, so they remain safe.

(2.4) In D''_T all upward/horizontal successors of "Initial" and "TobeEvaluated" attributes, have status "Initial" or "TobeEvaluated". This was the case in T. The only "TobeEvaluated" attribute that has been changed is *Att*. However *Att* has no "Initial" or "TobeEvaluated" predecessor.
□

A.6.2 Store in a synthesized attribute

The algorithms STORE-SYN-ATT and STORE-INH-ATT of Figure 2.10 are applied only under special circumstances. We prove that under these circumstances they preserve safety of an attributed tree T.

Lemma A.22
Given

- Synatt is a synthesized attribute of T_{Syn}*, a (nested) subtree of a safely attributed tree* T*,*

- all predecessors of Synatt have property safe-c,

- Newvalue is a correct value for Synatt w.r.t. T *and its inherited attributes,*
then after application of STORE-SYN-ATT*(Newvalue,Synatt)*

-a- *T is safely attributed;*
-b- *all predecessors of Synatt have property safe-c.*

Proof

(1) First, *Newvalue* becomes the new attribute value of *Synatt*, and the status of *Synatt* becomes either "Unchanged" or "Changed". We show that after this T is safely attributed.

(1.1) *Synatt* is safe, with property *safe-c*: Its new status is "Unchanged" or "Changed". By assumption, its value is correct w.r.t. T. Since all its direct predecessors have property *safe-c*, their values are correct w.r.t. T as well. Consequently, *Synatt* is correct w.r.t. its predecessors.

(1.2) The successors of *Synatt* are safe: Let *Succatt* be a successor of *Synatt*. We distinguish four cases according to the previous safety property of *Succatt*.

(1.2.(*safe-a*)) The status of *Succatt* remains "Initial". *Succatt* keeps property *safe-a*.

(1.2.(*safe-b*)) The status of *Succatt* remains "TobeEvaluated". Its value was correct w.r.t its direct predecessors with status "TobeEvaluated", "Initial" and "Unchanged". The value of at most one predecessor has changed (namely *Synatt*), but in that case the status of that predecessor has been set to "Changed". So *Succatt* still has property *safe-b*.

(1.2.(*safe-c*)) The status of *Succatt* remains "Unchanged" or "Changed". *Succatt* remains correct w.r.t. T because neither T nor any inherited attribute of the top of T has changed.

We prove that *Succatt* keeps property *safe-c*, by showing that *Succatt* remain correct w.r.t. its direct predecessors: The only predecessor whose status may have been changed is *Synatt*.

> (1.2.(*safe-c*).1) If *Synatt* had status "Unchanged" or "Changed" before application of STORE-SYN-ATT, it had property *safe-c* because by assumption, all its predecessors had property *safe-c*. Therefore its value was correct, so it was equal to *Newvalue*. Consequently the value of *Synatt* has not changed. Hence,*Succatt* remains correct w.r.t. its direct predecessors.
>
> (1.2.(*safe-c*).2) Otherwise *Synatt* had status "TobeEvaluated" or "Initial". Consequently, *Succatt* is not a direct successor of *Synatt*, because all direct successors of *Synatt* are either horizontal or upward successors, and since T was safe, these direct successors all had status "TobeEvaluated" or "Initial". Hence, *Succatt* remains correct w.r.t. its direct predecessors.

(1.2.(*safe-d*)) The status of *Succatt* remains "Unchanged" or "Changed".
Succatt is not a direct successor of *Synatt*, as mentioned already. Consequently *Succatt* remains correct w.r.t its direct predecessors.

Succatt has an inherited predecessor of an ancestor tree, with status "TobeEvaluated" or "Initial". No inherited attribute has been altered. So *Succatt* keeps property *safe-d*.

(1.3) Attributes that are not successors of *Synatt* are not affected. They remain safe.

(1.4) All upward and horizontal successors of "Initial" and "TobeEvaluated" attributes, have status "Initial" or "TobeEvaluated": This was the case before. The only "TobeEvaluated" attribute that may have been changed is *Synatt*. However all its predecessors of *Synatt* are "Unchanged" or "Changed", by assumption.

(2) Next, after storing *Newvalue* and resetting the status of *Synatt*, PROPAGATE-UNCHANGED is applied if the new status of *Synatt* is "Unchanged". According to Lemma A.21, T remains safely attributed after application of PROPAGATE- UNCHANGED.

(3) Finally, all predecessors of *Synatt* had property *safe-c*. the dependency graph D_T is cycle free. So no predecessor of *Synatt* is a successor of *Synatt*. STORE-SYNATT has only changed the status of successors of *Synatt*. So, the status of the predecessors of *Synatt* remains "Unchanged" or "Changed". The value of none of the predecessors of *Synatt* has changed, nor the value of their predecessors has changed. So, they remain correct w.r.t. their predecessors. Their values remain correct w.r.t. T because neither T nor any of its inherited attributes has been altered.

Hence, all predecessors of *Synatt* still have property *safe-c*.

□

A.6.3 Store in an inherited attribute

Lemma A.23

Given

- Inhatt is a inherited attribute of T_{Inh}, a (nested) subtree of a safely attributed tree T,

- all predecessors of Inhatt have property safe-c,

- Newvalue is a correct value for Inhatt w.r.t. T and its inherited attributes,

then after application of STORE-INH-ATT*(Newvalue,Inhatt)*

-a- *T is safely attributed;*

-b- *all predecessors of Inhatt have property safe-c.*

Proof

(1) Assume *Newvalue* equals the previous value of *Inhatt*.

(1.1) First, the status of *Inhatt* becomes "Unchanged". We prove that after this T is safely attributed.

(1.1.1) *Inhatt* is safe with property *safe-c*: *Inhatt* has status "Unchanged". Its value is correct w.r.t. T. Since we assumed that all predecessors had property *safe-c*, the value of all predecessors is correct w.r.t. T. Conseqently, *Inhatt* is correct w.r.t. its direct predecessors.

(1.1.2) Attributes that are not successors of *Inhatt* are not affected. They remain safe.

(1.1.3) All successors of *Inhatt* are safe: We distinguish them according to previous safety property.

(1.1.3.(*safe-a*)) Those that had property *safe-a*, still have status "Initial", and therefore they still have property *safe-a*.

(1.1.3.(*safe-b*)) These attributes still have status "TobeEvaluated". Their value was correct with respect to their direct "Initial", "TobeEvaluated" and "Unchanged" predecessors. No attribute value has changed. Therefore these attributes still have property *safe-b*.

(1.1.3.(*safe-c*)) These attributes still have status "Unchanged" or "Changed". Their value remains correct w.r.t. their direct predecessors, because no attribute value has changed. Their value remains correct w.r.t. T and the inherited attributes of T, beause neither of them has been altered. Hence these attributes still have property *safe-c*.

(1.1.3.(*safe-d*)) These attributes still have status "Unchanged" or "Changed". Their value remains correct w.r.t. their direct predecessors, because no attribute value has changed.

(1.1.3.(*safe-d*).1) For an attribute *Succatt outside* T_{Inh} of some tree T_{Succ} an inherited predecessor with status "Initial" or "TobeEvaluated" exists in the context tree in T of T_{Succ}. This inherited attribute also resides outside T_{Inh}, so it is not *Inhatt*. So the status of this inherited attribute remains "Initial" or "TobeEvaluated". Hence *Succatt* keeps property *safe-d*.

(1.1.3.(*safe-d*).2) An attribute *Succatt inside* T_{Inh} at some tree T_{Succ} has status "Unchanged" or "Changed", and its value is correct with respect to their direct predecessors.

(1.1.3.(*safe-d*).2.1) Assume *Succatt* still has an inherited predecessor at an ancestor node with status "TobeEvaluated" or "Initial", *Succatt* has property *safe-d*.

(1.1.3.(*safe-d*).2.2.) Assume *Succatt* has no inherited predecessor at an ancestor node with status "TobeEvaluated" or "Initial". We prove that *Succatt* now has property *safe-c*: We only need to prove that the value of its direct predecessors is correct.

(1.1.3.(*safe-d*).2.2.1) All direct *lower and horizontal* predecessors of *Succatt* have status "Unchanged" or "Changed" (Lemma A.14). None of these attributes has property *safe-d*, because if they would then *Succatt* would also have property *safe-d*, which would contradict our assumption. So, all lower and horizontal predecessors have property *safe-c*. Hence they have a correct value.

(1.1.3.(*safe-d*).2.2.2) All direct *upper* predecessors of *Succatt* are inherited attributes, at the parent node of the tree of *Succatt*. Since we assume that *Succatt* does not have property *safe-d*, the status of these inherited attributes is "Unchanged" or "Changed". (Lemma A.15). Moreover, if these inherited attributes would have property *safe-d* so would *Succatt*, hence they have property *safe-c*. Hence they have a correct value.

(1.1.4) All upward/horizontal successors of "TobeEvaluated" and "Initial" attributes still have status "TobeEvaluated" or "Initial". The only "TobeEvaluated" or "Initial" attribute whose status has changed is *Inhatt*. By assumption it had only "Unchanged" or "Changed" predecessors.

(1.2) Next, after resetting the status of *Synatt* to "Unchanged", the algorithm PROPAGATE-UNCHANGED is applied. According to Lemma A.21 T remains safely attributed after application of PROPAGATE-UNCHANGED.

(2) Assume *Newvalue* differs from the previous value of *Inhatt*. The status of *Inhatt* becomes "Changed", and PROPAGATE-TE-DOWN is applied.

(2.1) After this all attributes are safe in T:

(2.1.1) *Inhatt* now has status "Changed". Its value is correct w.r.t. T. Since we assumed that all predecessors had property *safe-c*, the value of all predecessors is correct w.r.t. T. Conseqently, *Inhatt* is correct w.r.t. its direct predecessors.

(2.1.2) Attributes that are not successors of *Inhatt* are not affected and remain safe.

(2.1.3) The successor attributes of *Inhatt* at the top of the direct subtrees T_i of T are safe:

After PROPAGATE-TE-DOWN has been applied they all have status "TobeEvaluated" or "Initial". PROPAGATE-TE-DOWN does not change the status of attributes with status "TobeEvaluated" or "Initial", nor does it proceed with their successors. However, it follows from the fact that T was safely attributed, that all these successors already have status "TobeEvaluated" or "Initial". We distinguish them according to previous safety property.

(2.1.3.(*safe-a*)) Those that had property *safe-a*, still have status "Initial", and therefore they still have property *safe-a*.

(2.1.3.(*safe-b*,*safe-c*,*safe-d*)) The new status of these attributes is "TobeEvaluated". The value of these attributes was correct w.r.t. their direct "Initial", "TobeEvaluated" and "Unchanged" predecessors. The value of at most one predecessor, namely *Inhatt*, has changed, but its status has changed to "Changed" as well. Therefore these attributes have property *safe-b*.

(2.1.4) The other successors of *Inhatt* in T_{Inh} are safe: let *Succatt* at some (nested) subtree T_{Succ} be such a successor. T_{Succ} is a subtree of a direct subtree T_i of T.

Succatt was safe in T_i because it was safe in T (Lemma A.16). *Succatt* is still safe in T_i because no attribute value in T_i has changed. We have proved that all inherited attributes of T_i are safe in T. According to Lemma A.17, *Succatt* is safe in T.

(2.1.5) The successor in the context tree of T_{Inh} are safe: Note that none of these successors is a direct successor of *Inhatt*. We distinguish four cases according to their previous safety property.

(2.1.5.(*safe-a*)) These attributes still have status "Initial". They still have property *safe-a*.

(2.1.5.(*safe-b*)) These attributes still have status "TobeEvaluated". Their value was correct with respect to their direct "Initial", "TobeEvaluated" and "Unchanged" predecessors. The value of at most one predecessor, namely *Inhatt*, has changed, but its status has changed to "Changed". Therefore these attributes still have property *safe-b*.

(2.1.5.(*safe-c*)) These attributes still have status "Unchanged" or "Changed". Their value remains correct w.r.t. their direct predecessors, because the value of none of these predecessors has changed. Their value remains correct w.r.t. T and its inherited attribute, because neither of them has been altered. Hence, these attributes still have property *safe-c*.

(2.1.5.(*safe-d*)) These attributes still have status "Unchanged" or "Changed". Their value remains correct w.r.t. their direct predecessors, because the value of none of these predecessors has changed.

These attributes have an inherited predecessor at an ancestor tree, with status "TobeEvaluated" or "Initial". Since they are successors of *Inhatt*, *Inhatt* is not one of these inherited predecessors. So these inherited predecessors still have status "Tobe-Evaluated" or "Initial". Hence these attributes still have property *safe-d*.

(2.2) All upward/horizontal successors of "TobeEvaluated" and "Initial" attributes have status "TobeEvaluated" or "Initial". Because this was the case before, and PROPAGATE-TE-DOWN has not changed the status of "TobeEvaluated" and "Initial" attributes. The upward/horizontal successors of the attributes that have been visited by PROPAGATE-TE-DOWN already have status "Initial" or "TobeEvaluated", because they are the upward/horizontal successors of the corresponding synthesized attributes of *Inhatt*.

(3) Finally, all predecessors of *Inhatt* had property *safe-c*. The dependency graph D_T is cycle free. So, no predecessor of *Inhatt* is a successor of *Inhatt*. STORE-SYNATT has only changed the status of successors of *Inhatt*. So, the status of the predecessors of *Inhatt* remains "Unchanged" or "Changed". The value of none of the predecessors of *Inhatt* has changed, nor the value of their predecessors has changed. So, they remain correct w.r.t. their predecessors. Their values remain correct w.r.t. T because neither T nor any of its inherited attributes has been altered. Hence, all predecessors of *Inhatt* still have property *safe-c*.

□

A.7 Reduction

The algorithm REDUCE-START of Figure 2.7 is invoked for incremental reduction of a term $\phi(T, t_1, \ldots, t_m)$. In this section we will prove that applying REDUCE-START in the presence of a safely attributed tree, results in a correct normal form for $\phi(T, t_1, \ldots, t_m)$ while T remains safely attributed.

Before we start this proof, we introduce an extra notion. Recall that we already introduced the notion of correct attributes in Section refsec.app.correct and the much weaker notion of a safe attribute in Section refsec.app.safe. When an attribute is inspected during reduction, we have to be sure that it either has a correct value or its status is "TobeEvaluated" or "Initial". So we need a notion that is weaker than "correct" and stronger than "safe":

Definition A.24 *An attribute Att in a safely attributed tree T with graph D_T is prepared for reduction if it has property safe-a or safe-b or safe-c.* □

Lemma A.25 *Attributes at the top node of a safely attributed tree are prepared for reduction.*

Proof The inherited attributes at a top node have no predecessors, so they cannot have property *safe-d*. Since they are safe, they have either property *safe-a*, *safe-b* or *safe-c*. □

REDUCE-START first invokes the algorithm REDUCE-PARS for reduction of the parameters of $\phi(T, t_1, \ldots, t_m)$. Then it invokes REDUCE-RHS. We prove in Section A.7.1 that after application of REDUCE-PARS, T is safely attributed and that predecessors of ϕ that will be inspected during reduction REDUCE-RHS are prepared for reduction. In Section A.7 we prove that the reduce function REDUCE that is used for further reduction by REDUCE-RHS preserves safety and returns correct normal forms, if all attributes are prepared for reduction when they are inspected.

When proving that the correct normal form is computed we assume that non-incremental reduction of a term as performed by the algorithm NON-INC-REDUCE always returns the proper normal form of that term.

A.7.1 Reduce parameters

We prove that application of REDUCE-PARS for inherited attributes at the top of a an attributed tree, will preserve safety of this attributed tree, and will result in correct normal forms for the parameter terms under consideration.

Lemma A.26
Given
- *T is a safely attributed tree,*
- *t_i is a parameter term of $\phi(T, t_1, \ldots, t_m)$ without any Φ-subterms,*
- *the inherited attribute associated with the i-th parameter of ϕ, $\mathsf{inh}(\phi, i)$, has status "Initial"*

then application of REDUCE-PAR*(t_i,$\mathsf{inh}(\phi, i)$,T) yields*
-a- *a correct normal form for t_i;*
-b- *a correct attribute value for $\mathsf{inh}(\phi, i)$;*
-c- *a safely attributed tree T.*

Proof
The status of $\mathsf{inh}(\phi, i)$ is "Initial". The algorithm REDUCE-PAR first invokes REDUCE (t_i, $\mathsf{inh}(\phi, i)$). Since t_i has no Φ-subterms, further reduction of t_i is by application of NON-INC-REDUCE.

-a- By assumption, this computes the correct normal form.
Next, REDUCE-PAR invokes STORE-INH-ATT.
-b- $\mathsf{inh}(\phi, i)$ is the inherited attribute at the top, so any value is correct.

-c- According to Lemma A.23, STORE-INH-ATT preserves safety.
□

Lemma A.27
Given
- T *is a safely attributed tree,*
- $\phi(T, t_i, \ldots, t_m)$ *is a* Φ*-term without* Φ*-subterms,*
- *all inherited attributes* $\mathsf{inh}(\phi, i)$ $(1 \le i \le m)$ *have status "Initial",*

then after application of REDUCE-PARS$(\phi(T, t_1, \ldots, t_m), T, T)$

-a- *the correct normal forms for all* t_i *have been yielded;*
-b- *all* $\mathsf{inh}(\psi, i)$ $(1 \le i \le m)$ *have correct attribute values;*
-c- T *is safely attributed;*
-d- ϕ *as well as its predecessors at the topnodes of direct subtrees* T_i *of* T *are prepared for reduction.*

Proof
(**-a-**,**-b-**,**-c-**) REDUCE-PARS$(\phi(T, t_1, \ldots, t_m), T, T)$ invokes REDUCE-PAR for each t_i. **-a-**, **-b-** and **-c-** follow directly from the previous lemma.

(**-d-**.1) ϕ is prepared for reduction because it is an attribute at a topnode of a safely attributed tree.

(**-d-**.2) All predecessors for ϕ at the top nodes of subtrees of T, are safe: Let *Predatt* be such a predecessor at subtree T_i. *Predatt* can not have property *safe-d*. Because the only inherited predecessors of *Predatt* at an ancestor node, are a subset of the inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, m)$. All these attributes have as a consequence of successive application of REDUCE-PAR, status "Unchanged" or "Changed". Hence, *Predatt* is prepared for reduction.
□

A.7.2 Reduce

In proving that application of REDUCE-START of Figure 2.7 returns the correct value. We make use of a similar result for REDUCE.

Theorem A.28
Given
- T *is a safely attributed tree,*
- τ *is a term with* Φ*-subterms over direct subtrees* T_i *of* T*,*
- *all* Φ*-subterms of* τ *are predecessors of an attribute* ϕ *of* T*,*
- *all attributes associated with* Φ*-subterms of* τ *are prepared for reduction,*

application of REDUCE(τ, T) *yields*

-a- *a correct normal form for* τ*;*
-b- *all inherited and synthesized attributes associated with the* Φ*-subterms in* τ *have property safe-c;*
-c- *a safely attributed tree* T*.*

Proof We prove the theorem by induction over the depth of T.

(T has depth 1) T has no subtrees. According to property (iii) in Section 2.3, τ contains no Φ-subterms. Reduction of τ is therefore by means of NON-INC-REDUCE. By assumption, NON-INC-REDUCE returns the correct normal form for τ (**-a-**). No attributes are associated with τ (**-b-**). Nothing in T has changed during reduction of τ. Hence, T remains safely attributed (**-c-**).

(T has depth N) Assume the theorem has been proved for any tree with depth $\leq N-1$. We proceed by structural induction over τ.

(T has depth N and τ *is a constant*)
In particular τ contains no Φ-subterms. So this case is equal to the case that T has depth 1.

(T has depth N and τ *has K function symbols*)
Assume we have proved **-a-**,**-b-** and **-c-** for any τ with less than K function symbols.
($N.K.1$) Assume τ is *not* a Φ-term: $f(s_1, \ldots, s_k)$. REDUCE is invoked to reduce the subterms of τ. These subterms contain less than K function symbols. By induction assumption, REDUCE returns a correct normal form for each s_i, while all attributes associated with Φ-subterms of s_i get property *safe-c*, and the attributed tree T remains safely attributed. The resulting normal forms are replaced in $f(s_1, \ldots, s_k)$, yielding $f(s'_1, \ldots, s'_k)$. $f(s'_1, \ldots, s'_k)$ contains no Φ-subterms, hence it is correctly reduced by NON-INC-REDUCE (**-a-**). Nothing changes in this last step. All attributes associated with Φ-terms in τ keep property *safe-c*, and T remains safely attributed (**-b-, -c-**).
($N.K.2$) Assume τ is a Φ-term $\psi(T_i, v_1, \ldots, v_k)$. The attributes ψ and $\mathsf{inh}(\psi, 1)$, $\ldots, \mathsf{inh}(\psi, k)$ of T_i are, by assumption, prepared for reduction in T.
($N.K.2.1$) Assume the status of ψ is "Unchanged" or "Changed". $\psi(T_i, v_1, \ldots, v_k)$ is replaced by the attribute value of ψ.
-b- ψ has property *safe-c* because it is prepared for reduction. Hence its value is correct w.r.t T and the inherited attributes $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, m)$ of T . All attributes associated with Φ-subterms in each v_j $(1 \leq j \leq k)$, as well as the inherited attributes $\mathsf{inh}(\psi, 1), \ldots, \mathsf{inh}(\psi, k)$ are horizontal predecessors of ψ. According to Lemma A.14, they all have status "Unchanged" or "Changed". According to the same line of reasoning, they have a correct value as well.
-a- By Definition A.4 of correct, the value of ψ is the correct normal form of $\psi(T_i, v_1, \ldots, v_k)$.
-c- Nothing has changed in T, so T remains safely attributed.
($N.K.2.2$) Assume the status of ψ is "TobeEvaluated" or "Initial".
($N.K.2.2.1$) First, REDUCE-PARS is applied to reduce the parameter terms $v_1, \ldots,$ v_k. For each i, $(1 \leq i \leq k)$, REDUCE-PAR$(v_i, \mathsf{inh}(\psi, i), T)$ is applied.

($N.K.2.2.1.1$) Assume the status of $\mathsf{inh}(\psi, i)$ is "Unchanged" or "Changed".
-b- $\mathsf{inh}(\psi, i)$ contains a correct value, hence it has property *safe-c*.
-a- By Definition A.4 of correct, this value is the proper normal form of v_i.
-c- Nothing has changed, so T remains safely attributed.

$(N.K.2.2.1.2)$ Assume the status of $\mathsf{inh}(\psi, i)$ is "TobeEvaluated" or "Initial".

$(N.K.2.2.1.2.1)$ First, REDUCE(v_i, T) is applied. Since v_i contains less than K function symbols, the induction assumption states that after this
-a- a correct normal form for v_i is obtained.
-b- attributes for all Φ-terms in v_i have obtained property *safe-c*.
-c- T is safely attributed (**-c-**).

$(N.K.2.2.1.2.2)$ Next, STORE-INH-ATT$(v_i, \mathsf{inh}(\psi, i))$ is applied.
-a- We already stated that a correct normal form for v_i has been obtained.
-b- By Definition A.4 of correct, the normal form of v_i is a correct attribute value for $\mathsf{inh}(\psi, i)$.
-c- After reduction of v_i, all direct predecessors of $\mathsf{inh}(\psi, i)$ have property *safe-c*. According to Theorem A.23, after application of STORE-INH-ATT: T remains safely attributed.

$(N.K.2.2.2)$ Next, REDUCE-RHS is applied to $\psi(T_i, v'_1, \ldots, v'_k)$, with v'_i the normal form of v_i.

$(N.K.2.2.2.1)$ Assume the status of ψ is "Unchanged" or "Changed".

-c- Nothing has changed in T, so T remains safely attributed.

-b- Since the attribute is prepared for reduction, its value is correct w.r.t. T and its predecessor $\mathsf{inh}(\phi, 1), \ldots, \mathsf{inh}(\phi, m)$ at the top of T.

-a- By Definition A.4 of correct, the attribute value is the correct normal form of $\psi(T_i, v'_1, \ldots, v'_k)$.

$(N.K.2.2.2.2)$ Assume the status of ψ is "TobeEvaluated" or "Initial".

$(N.K.2.2.2.2.1)$ First, REDUCE$(\psi(T_i, v'_1, \ldots, v'_k), T_i)$ is invoked. The depth of T_i is $< N$. By induction assumption
-a- REDUCE returns the correct normal form τ'.
-b- The new value of attribute ψ is correct w.r.t T.
-c- T is safely attributed.

$(N.K.2.2.2.2.2)$ Next, STORE-SYN(τ', ψ) is applied. Theorem A.22 can be used: because all direct predecessors of ψ have status "Unchanged" or "Changed", hence they have property *safe-c*.
-a- The normal form τ' does not change during STORE-SYN.
-b-, -c- STORE-SYN(τ', ψ) results in a correct attribute value for ψ, Theorem A.22 states that after application of the direct predecessors still have property *safe-c*. and T remains safely attributed.

□

A.7.3 Reduce-start

Finally, we can prove that application of REDUCE-START to a Φ-term returns a correct normal form, and preserves safety of the attributed tree.

Theorem A.29
Given
- *T is a safely attributed tree,*
- *$\phi(T, t_i, \ldots, t_m)$ is a Φ-term,*
- *all inherited attributes* $\mathsf{inh}(\phi, i)$ *$(i \leq i \leq m)$ have status "Initial",*

then application of REDUCE-START*$(\phi(T, t_i, \ldots, t_m), T)$ yields*

-a- *a correct normal form for $\phi(T, t_i, \ldots, t_m)$;*

-b- *a safely attributed tree T.*

Proof

(1) Assume the status of ϕ is "Unchanged" or "Changed". This can only be the case if no parameters $t_i, \ldots, t_m$ are present. The value of ϕ is returned as the normal form of $\phi(T)$.

T is safely attributed, so ϕ is safe. ϕ cannot have property *safe-d*, because it is an attributed at the top of T. So, ϕ has property *safe-c*. Hence, the value of ϕ is correct w.r.t T.

-a- By Definition A.4 of correct, this value is the correct normal form of $\phi(T)$

-b- Nothing has changed T remains safely attributed.

(2) Assume the status of ϕ is "TobeEvaluated" or "Initial".

(2.1) First, REDUCE-PARS is applied. According to Lemma A.27, this preserves safety of T. Moreover, all predecessor attributes of ϕ at subtrees T_i of T, are prepared for reduction.

(2.2) Next, REDUCE-RHS is applied.

(2.2.1) Assume the status of ϕ is "Unchanged" or "Changed". The value of ϕ is returned. ϕ is safe, and it is the attribute of the top of T. So, ϕ has property *safe-c*. Hence ϕ is correct. By Definition A.4 of correct

-a- $value_\phi$ is the correct normal form of $\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots, value_{\mathsf{inh}}(\phi, 1))$. Hence it is the correct normal form of $\phi(T, t_i, \ldots, t_m)$.

-b- Nothing has changed T remains safely attributed.

(2.2.2) Assume the status of ϕ is "TobeEvaluated" or "Initial".

$\phi(T, value_{\mathsf{inh}}(\phi, 1), \ldots, value_{\mathsf{inh}}(\phi, 1))$ matches the left-hand side of defining equation $eq_{\phi,p}$. Its right-hand side τ is instantiated. All attributes associated with Φ-subterms in τ are predecessors of ϕ at the subtrees T_i of T (Lemma A.1). All these attributes are prepared for reduction.

-a- According to Theorem A.28, application of REDUCE(τ, T) returns a correct normal form of τ.

Moreover, T remains safely attributed with a dependency graph D''_T, in which all attributes associated with τ have property *safe-c*. All inherited attributes $\mathsf{inh}(\phi, 1), \ldots,$

$\mathsf{inh}(\phi, m)$ have property *safe-c* as well. So, all direct predecessors of ϕ have property *safe-c*.

-b- According to Theorem A.22, subsequent application of STORE-SYN-ATT(τ', ϕ) preserves safety of T.

□

A.8 Reset Changed

The algorithm RESET-CHANGED of Figure 2.14 is applied to the attributed tree T, after a Φ-term $\phi(T, t_1, \ldots, t_m)$ has been reduced incrementally. RESET-CHANGED resets the status all "Changed" attributes to "Unchanged", The status of all "TobeEvaluated" successors of these "Changed" attributes is reset to "Initial". We prove that given an safely attributed tree, RESET-CHANGED returns a safely attributed tree, without attributes with status "Changed".

Theorem A.30 *Let T be a safely attributed tree. After* RESET-CHANGED*(T) has been applied T is still safely attributed, and no attributes in T have status "Changed".*

Proof

(1) We proof that T is safely attributed.

(1.1) All attributes in D_T are safe. We classify them according to their previous safety property.

(1.1.(*safe-a*)) Attributes with status "Initial" have not been changed by RESET-CHANGED so they still have status "Initial". Hence they still have property *safe-a*.

(1.1.(*safe-b*)) Attributes with status "TobeEvaluated", were correct w.r.t. their direct "TobeEvaluated", "Initial" and "Unchanged" predecessors.

If a "TobeEvaluated" attribute had "Changed" predecessors, its status is reset to "Initial" and it has property *safe-a*.

Otherwise, the set of "TobeEvaluated", "Initial" and "Unchanged" predecessors has not changed. So it still has property *safe-b*.

(1.1.(*safe-c*)) Attributes with status "Unchanged" have not been changed by RESET-CHANGED so they still have status "Unchanged". Attributes with status "Changed" now have status "Unchanged". No attribute value has changed, nor has T, so the attributes are still correct w.r.t. their direct predecessors, and w.r.t. the tree. Hence they still have property *safe-c*.

(1.1.(*safe-d*)) Attributes with status "Unchanged" have not been changed by RESET-CHANGED so they still have status "Unchanged". Attributes with status "Changed" now have status "Unchanged". No attribute value has changed, nor has T, so the attributes are still correct w.r.t. their direct predecessors. The attribute have an inherited predecessors with status "TobeEvaluated" or "Initial". The status of "TobeEvaluated" predecessors may have been altered into "Initial". They still have property *safe-d*.

(1.2) The successor attributes of "TobeEvaluated" and "Initial" attributes are still "TobeEvaluated" or "Initial". This was the case before application of RESET-

CHANGED. The status of some "TobeEvaluated" attributes may have been altered into "Initial". This clearly does not change the property.

(2) Clearly no attributes in T have status "Changed" after application of RESET-CHANGED.

□

Appendix B

Layered PRSs and higher order attribute grammars

We present a translation of well-presented *layered* primitive recursive schemes as defined in Chapter 5, into strongly non-circular higher-order attribute grammars. And a translation of strongly non-circular higher-order attribute grammars into algebraic specifications.

Higher order attribute grammars (HAGs) have been designed to remove the separation between the syntax description and the semantic description in attribute grammars [VSK90, TC90]. To this aim so-called *Non-terminal attributes*, NTAs, can be added to an attribute grammar. An NTA is either a non-terminal of the grammar which is used as a value in an attribute equation, or an attribute value described by means of production rules. The latter kind can be added as extra arguments to constructors of the underlying grammar. The attribute rules for the constructor specify the value of the NTA itself, and of its inherited attributes.

In translating a layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ with sub PRS $\langle G_2, S_2, \Phi_2,$-$Eq_2, Eq_{\Phi_2}\rangle$ into a HAG, G_2-terms will be mapped onto non-terminal attributes.

Example B.1 When translating the type checker specification in Figure 5.1 and 5.4 the equation

$$\mathsf{tcstm}(\mathsf{assign}(\mathit{Id}, \mathit{Exp}), \mathit{Tenv}) = \mathsf{compatible}(\mathsf{lookup}(\mathit{Tenv}, \mathit{Id}), \mathsf{tcexp}(\mathit{Exp}, \mathit{Tenv}))$$

would cause the assign constructor to be extended with a non-terminal attribute $\overline{\mathsf{TENV}}$. lookup is the synthesized attribute of the type-environment. The value of the corresponding inherited attribute, $\mathsf{Id}_{\overline{\mathsf{TENV}}}$ is the subtree ID of the assign constructor. Hence, ID is also used as a non-terminal attribute. The resulting attribute rules are:

$$\mathsf{assign} : \mathsf{STM} \rightarrow \mathsf{ID} \times \mathsf{EXP} \times \overline{\mathsf{TENV}} \left\{ \begin{array}{l} \mathsf{tcstm}_{\mathsf{STM}} = \mathsf{compatible}(\mathsf{lookup}_{\overline{\mathsf{TENV}}}, \mathsf{tcexp}_{\mathsf{EXP}}) \\ \overline{\mathsf{TENV}} = \mathsf{Tenv}_{\mathsf{STM}} \\ \mathsf{Id}_{\overline{\mathsf{TENV}}} = \mathsf{ID} \\ \mathsf{Tenv}_{\mathsf{EXP}} = \mathsf{Tenv}_{\mathsf{STM}} \end{array} \right.$$

□

Definition B.1 A higher order attribute grammar is strongly non-circular if for each node in an (expanded) tree an attribute evaluation order exists that does not depend on the particular subtree rooted at that node. □

B.1 Layered PRS → strongly non-circular HAG

We translate the well-presented layered PRS $\langle G_1, S_1, \Phi_1, Eq_1, Eq_{\Phi_1}\rangle$ with a subPRS, $\langle G_2, S_2, \Phi_2, Eq_2, Eq_{\Phi_2}\rangle$, into a higher-order attribute grammar. The subPRS can be layered itself. A non-terminal attribute gets a $\overline{\text{bar}}$ when used as NTA in a production rule.

(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 (See [CFZ82] and Chapter 2): Synthesized attributes ϕ are associated with incremental functions ϕ and inherited attributes $\mathsf{inh}(\phi, \mathsf{k})$ with the parameters of these functions. Attribute rules for the constructors of G_1 are derived from the defining equations.

Example B.2 The following attribute rules are derived from the equation in Example B.1.

$$\mathsf{assign} : \mathsf{STM} \rightarrow \mathsf{ID} \times \mathsf{EXP} \left\{ \begin{array}{l} \mathsf{tcstm}_{\mathsf{STM}} = \mathsf{compatible}(\mathsf{lookup}(\mathsf{Tenv}_{\mathsf{STM}}, \mathsf{ID}), \mathsf{tcexp}_{\mathsf{EXP}}) \\ \mathsf{Tenv}_{\mathsf{EXP}} = \mathsf{Tenv}_{\mathsf{STM}} \end{array} \right.$$

□

(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 non-terminal 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 non-terminal 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 non-terminal 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 non-terminal attribute.

Example B.3 In Example B.1 lookup is a Φ_2-function. Therefore, $\overline{\mathsf{TENV}}$ is added to the constructor. The attributes rules are altered by adding $\overline{\mathsf{TENV}} = \mathsf{Tenv}_{\mathsf{STM}}$ (a) and $\mathsf{Id}_{\overline{\mathsf{TENV}}} = \mathsf{ID}$ (b), and by replacing $\mathsf{lookup}(\mathsf{Tenv}_{\mathsf{STM}}, \mathsf{ID})$ by $\mathsf{lookup}_{\overline{\mathsf{TENV}}}$ (c). As a result we obtain the attribute rules in Example 1. □

(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. For constructors without NTAs, merging two occurrences is simply taking the union of the attribute

rules. Merging two constructors $p : X_0 \rightarrow X_1 \times \ldots \times X_n \times \overline{NTA_1}$ and $p : X_0 \rightarrow X_1 \times \ldots \times X_n \times \overline{NTA_2}$ with attribute rules R_p and R'_p yields $p : X_0 \rightarrow X_1 \times \ldots \times X_n \times \overline{NTA_1} \times \overline{NTA_2}$ with rules $R_p \cup R'_p$.

The resulting HAG is strongly non-circular.

Step 1 generates a strongly non-circular AG, according to [CFZ82]. We check that step 2 preserves strong non-circularity for the subtrees and NTAs of each constructor p. Let ψ be the synthesized attribute at a subtree X_i or at an non-terminal attribute $\overline{NTA}$. The predecessors of this attribute at the same subtree are a subset of the inherited attributes $\mathsf{inh}(\psi, \mathsf{k})$. Cyclic dependencies are excluded and strong non-circularity is guaranteed, when the context tree $p(X_1, \ldots, X_i, \ldots, X_n)$ does not impose a dependency from ψ to $\mathsf{inh}(\psi, \mathsf{k})$. Attribute dependencies are derived from attribute rules, which are, in turn, derived from defining equations. We use properties of defining equations to proof that an attribute ψ never depends on an attribute $\mathsf{inh}(\psi, \mathsf{k})$ at the same tree.

For X_i this is the result of the well-presentedness property. Consider a defining equation over p like

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

A dependency from ψ to some $\mathsf{inh}(\psi, \mathsf{k})$ is derived from this equation only if $\psi(x_i, v_1, \ldots, v_k, \ldots, v_m)$ occurs in τ and parameter term v_k has a subterm $\psi(x_i, v'_1, \ldots, v'_m)$. This can only mean that v_k and v'_k are not equal, hence that the equation is not well-presented. The well-presentedness property states that in τ all occurrences of parameters of ϕ over the same x_i should be equal.

Well-presentedness does not apply to Φ_2-terms in the-right hand side of a Φ_1-defining equation. Suppose $\psi_2(Aux, w_1, \ldots, w_l)$ is a subterm of τ, then w_k itself can contain the subterm $\psi_2(Aux, w'_1, \ldots, w'_l)$. This does not introduce cyclic dependencies because each occurrence of a Φ_2-term yields a separate NTA. So, an inherited attribute $\mathsf{inh}(\psi_2, \mathsf{k})$ at an NTA will never depend on a synthesized attribute, ψ_2 or another one, of the same NTA.

Step 3 applies step 1 and 2 and yields attribute rules for a strongly non-circular HAG. Merging these rules preserves strong non-circularity.

B.2 Strongly non-circular HAG → algebraic specification

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

(1) Each synthesized attribute ϕ of a sort X is associated with a function ϕ_1. The first argument of that function is X. Parameters of this function are the inherited attributes of X on which the value of the synthesized attribute depends [CFZ82].

Example B.4 From the HAG in Example B.1 functions will be created for the type check attributes, and for the lookup attribute. □

(2) For each constructor

$$p : X_0 \rightarrow X_1 \times \ldots \times X_n \times \overline{NTA_1} \times \ldots \times \overline{NTA_k}$$

the constructor without its NTAs is added to the signature of the algebraic specification.

(3) Defining equations are derived from attribute rules of constructor p as follows. For each rule $\phi_{X_0} = \tau$ of p create an equation

$$\phi_1(p(x_1, \ldots, x_n, y_1, \ldots, y_m) = \tau$$

Until there is no attribute or $\overline{\mathrm{NTA}}$ left in τ do:

(a) Replace every inherited attribute of X_0 by the variable y_i of the associated parameter.

(b) Replace every synthesized attribute $\psi_{X_i}, i > 0$, by a term $\psi_1(x_i,\ z_1, \ldots, z_l)$, with z_j the inherited attributes that form the parameters of ψ.

(c) Replace every inherited attribute of X_i, $i > 0$, by the right-hand side of its defining rule.

(d) Replace every synthesized attribute $\chi_{\overline{\mathrm{NTA_i}}}$ by a term $\chi_2(\mathrm{NTA_i},\ w_1, \ldots, w_l)$, with w_j the inherited attributes that form the parameters of χ.

(e) Replace every $\overline{\mathrm{NTA_i}}$ 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. The constructor while is extended with a nonterminal attribute $\overline{\mathsf{STM}}$.

$$\mathsf{while} : \mathrm{STM} \rightarrow \mathrm{EXP} \times \mathrm{STMS} \times \overline{\mathsf{STM}} \left\{ \begin{array}{l} \overline{\mathsf{STM}} = \begin{array}{ll} \mathsf{if} & \mathsf{evalexp}_{\mathsf{EXP}} = \mathsf{true} \\ \mathsf{then} & \mathsf{while}(\mathsf{EXP}, \mathsf{STMS}) \\ \mathsf{else} & \mathsf{skip} \end{array} \\ \mathsf{Venv}_{\mathsf{EXP}} = \mathsf{Venv}_{\mathsf{STM}} \\ \mathsf{Venv}_{\mathsf{STMS}} = \mathsf{Venv}_{\mathsf{STM}} \\ \mathsf{Venv}_{\overline{\mathsf{STM}}} = \mathsf{evalstms}_{\mathsf{STMS}} \\ \mathsf{evalstm}_{\mathsf{STM}} = \mathsf{evalstm}_{\overline{\mathsf{STM}}} \end{array} \right.$$

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.

$$\frac{\mathsf{evalexp}(Exp,\, Venv) = \mathsf{true}}{\mathsf{evalstm}(\mathsf{while}(Exp, Stms),\, Venv) = \mathsf{evalstm}(\mathsf{while}(Exp, Stms), \mathsf{evalstms}(Stms,\, Venv))}$$

$$\frac{\mathsf{evalexp}(Exp,\, Venv) \neq \mathsf{true}}{\mathsf{evalstm}(\mathsf{while}(Exp, Stms),\, Venv) = \mathsf{evalstm}(\mathsf{skip}, \mathsf{eval}(Stms,\, Venv))}$$

Both equations are allowed in an algebraic specification but they 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 PRSs as defined by property (iii.a) in Definition 2.3 is violated.

Appendix C

Example specification

This appendix contains the ASF+SDF specification of the syntax and static semantics of the toy language Pico.

module Layout

exports

lexical syntax

"%%" ~[\n]* [\n] → LAYOUT
[\t\n] → LAYOUT

module Identifiers

imports Layout

exports

sorts ID

lexical syntax

[a-z] [a-z0-9]* → ID

variables

Id [0-9′]* → ID

module Strings

imports Layout

exports

sorts STR-CON STRING

lexical syntax

"\"" ~[\"\n]* "\"" → STR-CON

context-free functions

STR-CON → STRING
STRING "||" STRING → STRING {**left**}

variables

Str-con [0-9′]* → STR-CON
Str [0-9′]* → STRING

hiddens

variables

Chars [*12*] $\longrightarrow$ CHAR$*$

equations

[S1] str-con(“"” *Chars*$_1$ “"”) || str-con(“"” *Chars*$_2$ “"”) $=$
str-con(“"” *Chars*$_1$ *Chars*$_2$ “"”)

module Integers

exports

sorts NAT-CON DIGIT

lexical syntax

[0-9] $\longrightarrow$ DIGIT

DIGIT$+$ $\longrightarrow$ NAT-CON

variables

Nat-con [*0-9'*]$*$ $\longrightarrow$ NAT-CON

module Booleans

imports Layout

exports

sorts BOOL

context-free functions

true $\longrightarrow$ BOOL

false $\longrightarrow$ BOOL

BOOL “|” BOOL $\longrightarrow$ BOOL {**left**}

BOOL “&” BOOL $\longrightarrow$ BOOL {**left**}

“(” BOOL “)” $\longrightarrow$ BOOL {**bracket**}

variables

Bool [*0-9'*]$*$ $\longrightarrow$ BOOL

priorities

BOOL “|” BOOL $\longrightarrow$ BOOL $<$ BOOL “&” BOOL $\longrightarrow$ BOOL

equations

[Bo1] true | *Bool* $=$ true

[Bo2] false | *Bool* $=$ *Bool*

[Bo3] true & *Bool* $=$ *Bool*

[Bo6] false & *Bool* $=$ false

module Types

imports Booleans

exports

sorts TYPE

context-free functions

natural $\longrightarrow$ TYPE

string $\longrightarrow$ TYPE

compatible “(” TYPE “,” TYPE “)” $\longrightarrow$ BOOL

variables

Type [*0-9′*]∗ $\rightarrow$ TYPE

equations

[Typ1] compatible(natural, natural) = true

[Typ2] compatible(string, string) = true

module Type-environments

imports Identifiers Types

exports

sorts TENV PAIR

context-free functions

"(" ID ":" TYPE ")" $\rightarrow$ PAIR

"identifier" "of" PAIR $\rightarrow$ ID

"type" "of" PAIR $\rightarrow$ TYPE

empty-tenv $\rightarrow$ TENV

PAIR "+" TENV $\rightarrow$ TENV

lookup TENV for ID $\rightarrow$ TYPE

variables

Tenv [*0-9′*]∗ $\rightarrow$ TENV

Pair [*0-9*]∗ $\rightarrow$ PAIR

equations

$$[\text{Lkp1}]\quad \frac{\text{identifier of } \mathit{Pair} = \mathit{Id}}{\text{lookup } \mathit{Pair} + \mathit{Tenv} \text{ for } \mathit{Id} = \text{type of } \mathit{Pair}}$$

$$[\text{Lkp2}]\quad \frac{\text{identifier of } \mathit{Pair} \neq \mathit{Id}}{\text{lookup } \mathit{Pair} + \mathit{Tenv} \text{ for } \mathit{Id} = \text{lookup } \mathit{Tenv} \text{ for } \mathit{Id}}$$

[T1] identifier of $(\mathit{Id} : \mathit{Type}) = \mathit{Id}$

[T2] type of $(\mathit{Id} : \mathit{Type}) = \mathit{Type}$

module Pico-syntax

imports Identifiers Integers Types Strings

exports

sorts PROGRAM DECLS SERIES STATEMENT EXP
ID-TYPE-LIST ID-TYPE

context-free functions

```
begin DECLS SERIES end                         → PROGRAM

declare ID-TYPE-LIST ";"                       → DECLS
ID-TYPE "," ID-TYPE-LIST                       → ID-TYPE-LIST
ID-TYPE                                        → ID-TYPE-LIST
ID ":" TYPE                                    → ID-TYPE

STATEMENT ";" SERIES                           → SERIES
STATEMENT                                      → SERIES
ID ":=" EXP                                    → STATEMENT
"if" EXP "then" SERIES "else" SERIES "fi"      → STATEMENT
"while" EXP "do" SERIES "od"                   → STATEMENT
ID                                             → EXP
NAT-CON                                        → EXP
STR-CON                                        → EXP
EXP "+" EXP                                    → EXP {left}
EXP "−" EXP                                    → EXP {left}
EXP "||" EXP                                   → EXP {left}
"(" EXP ")"                                    → EXP {bracket}
```

variables

```
Decls [0-9']*       → DECLS
Id-type-list        → ID-TYPE-LIST
Id-type [0-9']*     → ID-TYPE
Series [0-9']*      → SERIES
Stat [0-9']*        → STATEMENT
Exp [0-9']*         → EXP
```

module Pico-typecheck

imports Booleans Pico-syntax Type-environments

exports

context-free functions

```
"tcp" PROGRAM                   → BOOL {incremental}
"tcd" DECLS                     → TENV {incremental}
"tcidlist" ID-TYPE-LIST in TENV → TENV {incremental}
"tcid" ID-TYPE                  → PAIR {incremental}
"tcs" SERIES in TENV            → BOOL {incremental}
"tcstat" STATEMENT in TENV      → BOOL {incremental}
"[" EXP "]" in TENV             → TYPE {incremental}
```

equations

[NITc1] tcp begin *Decls Series* end = tcs *Series* in tcd *Decls*

[NITc2] tcd declare *Id-type-list*; = tcidlist *Id-type-list* in empty-tenv

$$[\text{NITc3}]\quad \frac{\text{tcidlist } \textit{Id-type-list} \text{ in } \textit{Pair-list} = \textit{Pair-list}_1}{\text{tcidlist } \textit{Id-type}, \textit{Id-type-list} \text{ in } \textit{Pair-list} = \text{tcid } \textit{Id-type} + \textit{Pair-list}_1}$$

[NITc3a] tcidlist *Id-type* in *Pair-list* = tcid *Id-type* + *Pair-list*

[NITc4] tcid *Id* : *Type* = (*Id* : *Type*)

[NITc5] tcs *Stat*; *Series* in *Tenv* = tcstat *Stat* in *Tenv* & tcs *Series* in *Tenv*

[NITc5a] tcs *Stat* in *Tenv* = tcstat *Stat* in *Tenv*

[NITc6a] tcstat *Id* := *Exp* in *Tenv* = compatible(lookup *Tenv* for *Id*, [*Exp*] in *Tenv*)

$$[\text{NITc6b}]\quad \frac{\begin{array}{c}[\textit{Exp}] \text{ in } \textit{Tenv} = \text{natural},\\ \text{tcs } \textit{Series}_1 \text{ in } \textit{Tenv} = \textit{Bool}_1,\\ \text{tcs } \textit{Series}_2 \text{ in } \textit{Tenv} = \textit{Bool}_2\end{array}}{\text{tcstat if } \textit{Exp} \text{ then } \textit{Series}_1 \text{ else } \textit{Series}_2 \text{ fi in } \textit{Tenv} = \textit{Bool}_1 \;\&\; \textit{Bool}_2}$$

$$[\text{NITc6c}]\quad \frac{\begin{array}{c}[\textit{Exp}] \text{ in } \textit{Tenv} = \text{natural},\\ \text{tcs } \textit{Series}_1 \text{ in } \textit{Tenv} = \textit{Bool}\end{array}}{\text{tcstat while } \textit{Exp} \text{ do } \textit{Series}_1 \text{ od in } \textit{Tenv} = \textit{Bool}}$$

[NITc7a] [*Id*] in *Tenv* = lookup *Tenv* for *Id*

[NITc7b] [*Nat-con*] in *Tenv* = natural

[NITc7c] [*Str-con*] in *Tenv* = string

$$[\text{NITc8a}]\quad \frac{\begin{array}{c}[\textit{Exp}_1] \text{ in } \textit{Tenv} = \text{natural},\\ [\textit{Exp}_2] \text{ in } \textit{Tenv} = \text{natural}\end{array}}{[\textit{Exp}_1 + \textit{Exp}_2] \text{ in } \textit{Tenv} = \text{natural}}$$

$$[\text{NITc8b}]\quad \frac{\begin{array}{c}[\textit{Exp}_1] \text{ in } \textit{Tenv} = \text{natural},\\ [\textit{Exp}_2] \text{ in } \textit{Tenv} = \text{natural}\end{array}}{[\textit{Exp}_1 - \textit{Exp}_2] \text{ in } \textit{Tenv} = \text{natural}}$$

$$[\text{NITc8c}]\quad \frac{\begin{array}{c}[\textit{Exp}_1] \text{ in } \textit{Tenv} = \text{string},\\ [\textit{Exp}_2] \text{ in } \textit{Tenv} = \text{string}\end{array}}{[\textit{Exp}_1 \,||\, \textit{Exp}_2] \text{ in } \textit{Tenv} = \text{string}}$$

Bibliography

[AC90] I. Attali and J. Chazarain. Functional evaluation of strongly non circular typol specifications. In P.Deransart M. Jourdan, editor, *Attribute grammars and their applications - Proceedings of the WAGA conference*, volume 461 of *Lecture Notes in Computer Science*, pages 157–176. Springer-Verlag, 1990.

[ACG92] I. Attali, J. Chazarain, and S. Gilette. Incremental evaluation of natural semantics specifications. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming, '92*, volume 631 of *Lecture Notes in Computer Science*, pages 87–99. Springer-Verlag, 1992.

[AFZ88] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Logic Programming Series, pages 160–177. MIT Press, 1988.

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

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).

[BHK87] J.A. Bergstra, J. Heering, and P. Klint. ASF - an algebraic specification formalism. Report CS-R8705, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1987.

[BHK89a] 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.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[Bir80] R.S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, 1980.

[Bra90] M.G.J. van den Brand. Incremental affix evaluation in syntax directed editors. In *Conference Proceedings of Computing Science in the Netherlands, CSN'90*, pages 35–49. SION, 1990.

[Bra92] M.G.J. van den Brand. *Pregmatic, A generator for incremental programming environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[Bra93] M.G.J. van den Brand. Prettyprinting without losing comments. Report P9315, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Bra93.ps.Z.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[CDD+85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Rapports de Recherche 416, INRIA, Sophia Antipolis, 1985.

[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.

[Des88] Th. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands CSN'91*, pages 150-164. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Deu91.ps.Z.

[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.

[DM93] A. van Deursen and P.D. Mosses. Executing Action Semantics descriptions using ASF+SDF. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, AMAST93*, Workshops in Computing, pages 415–416. Springer-Verlag, 1993. System demonstration.

[DT89] Th. Despeyroux and L. Théry. *TYPOL - User's guide and manual.* INRIA, Sophia-Antipolis, 1989.

[Fie92] J. Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, 1992.

[Fie93] J. Field. A graph reduction approach to incremental rewriting. In C. Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 259–273, 1993.

[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).

[Hen89a] P.R.H. Hendriks. Lists and associative functions in algebraic specifications - semantics and implementation. Report CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.

[Hen89b] P.R.H. Hendriks. Typechecking Mini-ML. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 299–337. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 7.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[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.

[HS84] W. Henhapl and G. Snelting. Context relations - a concept for incremental context analysis in program fragments. In *Programmiersprachen und Programmentwicklung, 8. GI Fachtagung*, volume 77 of *Informatik-Fachberichte*. Springer-Verlag, 1984.

[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.

[Hud91] S.E. Hudson. Incremental attribute evaluation: a flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.

[JBP90] M. Jourdan, C. Le Bellec, and D. Parigot. The Olga attribute grammar description language: design, implementation and evaluation. In P.Deransart M. Jourdan, editor, *Attribute grammars and their applications - Proceedings of the WAGA conference*, volume 461 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, 1990.

[Jeu91] J. Jeuring. Incremental algorithms on lists. In J. van Leeuwen, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'91*, pages 315–335. SION, 1991.

[Jou84] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 81–93. ACM, 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.

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[Kas84] U. Kastens. The GAG-System - A tool for compiler construction. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 165–181. Cambridge University Press, 1984.

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

[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.

[Kli93a] P. Klint, editor. *The ASF+SDF Meta-environment User's Guide*, version 26 February. 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as SysManual.ps.Z.

[Kli93b] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Koo92a] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92a.ps.Z.

[Koo92b] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Appeared as Report P9202, University of Amsterdam. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92b.ps.Z.

[Kos91] C.H.A Koster. Affix grammars for programming languages. In H. Alblas and B. Melichar, editors, *International Summer School on Atribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, 1991.

[KP90] K. Koskimies and J. Paakki. *Automating Language Implementation - a pragmatic approach.* Ellis Horwood books in information technology. Ellis Horwood, 1990.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[LeL90] INRIA, Rocquencourt. *LeLisp, Version 15.23, reference manual*, 1990.

[MBD+90] B. Magnusson, M. Begtsson, L.O Dahlin, G. Fries, A. Gustavson, G. Hedin, S. Minör, D. Oscarsson, and M. Taube. An overvieuw of the mjølner/orm environment: Incremental language and software development. Technical Report LU-CS-TR:90-57 and LUTEDX/(TECS-3026)/1-12/(1990), Lund University and Lund Institute of Technology, 1990.

[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.

[Meu92] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 277–286. Springer-Verlag, 1992. Full version available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu90.ps.Z.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[Mos92] P.D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Pai86] R. Paige. Programming with invariants. *IEEE Sofware*, 3(1):56–69, 1986.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[PSV92] M. Pennings, D. Swierstra, and H. Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming '92*, volume 631 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, 1992.

[PT89] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM, 1989.

[Rep82] T. Reps. Generating language-based environments. Technical report TR 82-514, Cornell University, Ithaca, 1982. Ph.D. Thesis.

[RMT86] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language-based editors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–13. ACM, 1986.

[RR93] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 502–510. ACM, 1993.

[RT89a] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[RT89b] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual - Third edition*. 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):449–477, 1983.

[Sne91] G. Snelting. The calculus of context relations. *Acta Informatica*, 28:411–445, 1991.

[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. Appeared as *SIGPLAN Notices* 25(6).

[Vog93] H.H. Vogt. *Higher Order Attribute Grammars*. PhD thesis, Rijksuniversiteit Utrecht, 1993.

[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.

Samenvatting

Het werk dat beschreven is in dit proefschrift is uitgevoerd in het kader van het ESPRIT[1] project GIPE (Generation of Interactive Programming environments) en het NWO project Incrementele programmageneratoren. Binnen deze projecten wordt gewerkt aan de ontwikkeling van een generator van programmeeromgevingen: het ASF+SDF-systeem.

In deze samenvatting leg ik eerst uit wat een generator van programmeeromgevingen is, wat incrementele technieken zijn en waarom die gewenst zijn in programmeeromgevingen. Daarna schets ik de incrementele techniek die het onderwerp is van dit proefschrift.

Een generator van programmeeromgevingen

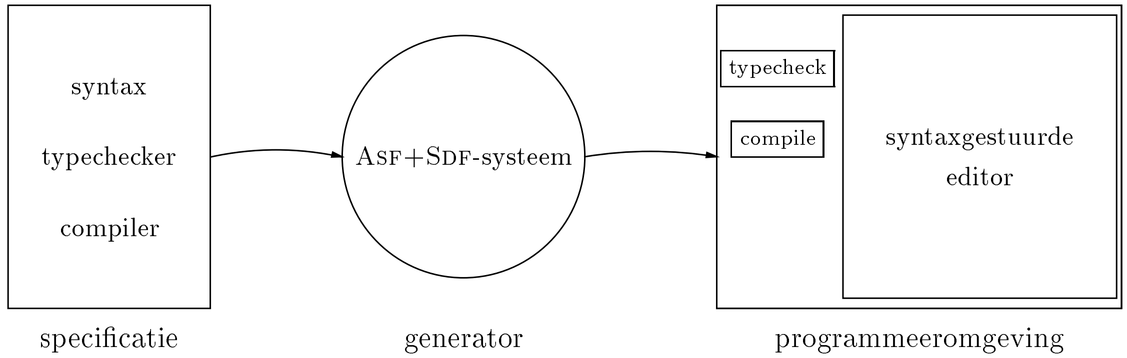
Onder een programmeeromgeving van een (programmeer) taal verstaan we de verzameling van alle hulpmiddelen (*tools*) die gebruikt kunnen worden om programma's in die taal te herkennen, controleren, analyseren vertalen of uit te voeren. Voor bekende programmeertalen als bijvoorbeeld Pascal en Basic zijn omgevingen beschikbaar voor de meeste soorten computers. Voor nieuw te ontwikkelen programmeertalen kan een generator van programmeeromgevingen uitkomst bieden. Die creëert een omgeving op basis van een formele beschrijving van de syntax van die taal en van de gewenste hulpmiddelen.

Zo'n gegenereerde omgeving bestaat in ieder geval uit een syntaxgestuurde editor en bevat meestal een typechecker. Een syntaxgestuurde editor is een editor die controleert of de tekst die wordt ingetypt voldoet aan de syntax van de gedefinieerde taal. Een typechecker voert extra controles uit op syntactisch correcte programma's, bijvoorbeeld of gebruikte variabelen gedeclareerd zijn en of ze op de juiste manier gebruikt worden.

In het plaatje op pagina 162 wordt een programmeeromgeving weergegeven als een *window* met daarbinnen een syntaxgestuurde editor. Het window is uitgerust met knoppen waarop gedrukt kan worden als van een programma in de editor de types gecontroleerd moeten worden of wanneer het of vertaald moet worden naar een programma in machinetaal.

Het is prettig als het controleren van een programma snel gaat, zodat tijdens het intypen regelmatig kan worden gekeken of alles nog klopt. Om die reden zijn

[1] European Strategic Programme for Research in Information Technology



de syntaxcheckers in de editors die door het ASF+SDF-systeem worden gegenereerd *incrementeel*. Dit betekent dat wanneer in een programma iets is veranderd niet het hele programma opnieuw wordt gecontroleerd maar alleen het nieuwe gedeelte. In het algemeen noemen we een methode incrementeel als gebruik wordt gemaakt van resultaten van een eerdere berekening om een nieuwe resultaat zo snel mogelijk te verkrijgen.

Het doel van het onderzoek in dit proefschrift is om ook typecheckers en eventueel andere gereedschappen die door het ASF+SDF-systeem zijn gegenereerd incrementeel te maken.

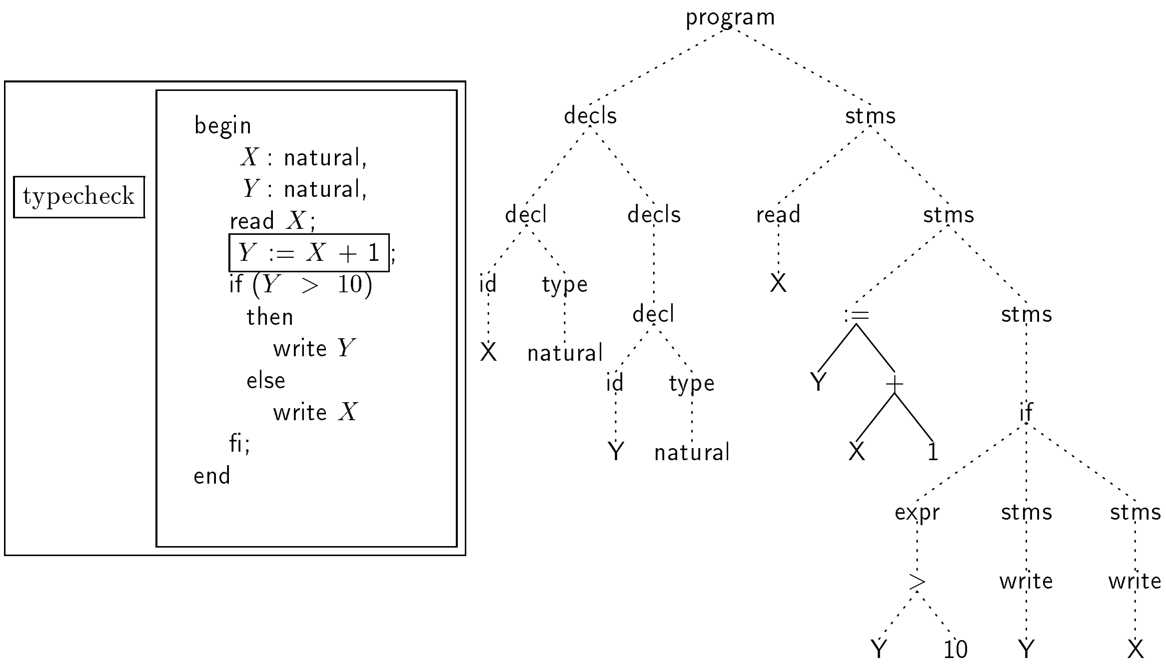
Incrementeel syntaxchecken

De specificatie van de syntax van een taal wordt gebruikt door een syntaxgestuurde editor. De syntaxchecker van een editor probeert de tekst in de editor te ontleden. Als de tekst een syntactisch correct programma is kan de de structuur van het programma worden weergegeven in de vorm van een *abstracte syntax boom*. Wanneer in het programma iets wordt veranderd nadat de syntax is gecontroleerd hoeft niet het hele programma opnieuw ontleed te worden maar alleen het nieuwe stuk. De abstracte syntax boom van het gewijzigde programma wordt dan *incrementeel* samengesteld. Op de bladzij hiernaast staat een voorbeeld van een programma en zijn abstracte syntax boom. Als het statement $\boxed{\mathsf{Y} := \mathsf{X} + 1}$ wordt gewijzigd hoeft alleen het stukje boom zonder stippellijntjes te worden vervangen.

Typechecken

Een typechecker controleert een syntactisch correct programma verder. Er wordt bijvoorbeeld gekeken of variabelen die in de statements staan wel gedeclareerd zijn en of expressies juist getypeerd zijn.

Het programma in het voorbeeld is goed getypeerd. De variabelen Y en X zijn gedeclareerd. Omdat ze allebei type natural hebben is het statement Y := X + 1 goed en mag Y met 10 worden vergeleken in Y > 10. Y := X + "aap" zou fout zijn want het woord "aap" kan niet bij een getal worden opgeteld. Ook als Y := X + 1 wordt



veranderd in Y := Z − 9 is dat statement fout getypeerd. De variabele Z is immers niet gedeclareerd.

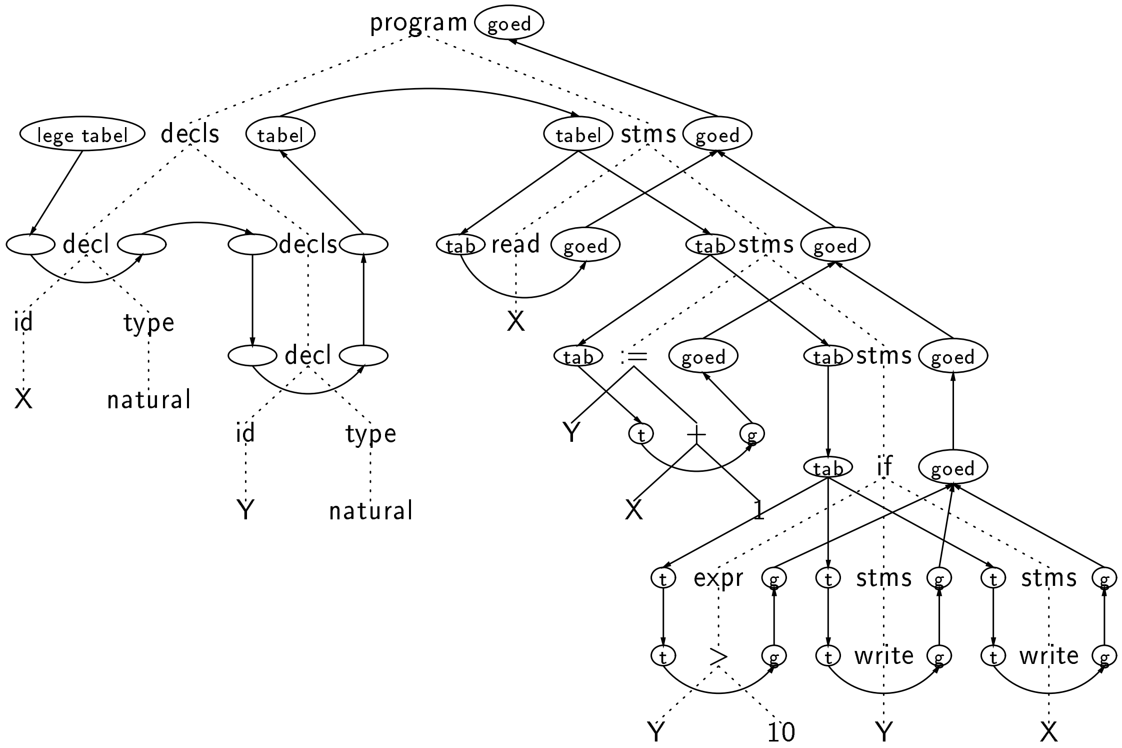
Een typecontrole wordt meestal in twee fasen uitgevoerd. Eerst worden de declaraties gecontroleerd waarbij de informatie over de variabelen in een tabel wordt verzameld. Vervolgens worden aan de hand van die tabel de statements één voor één gecontroleerd.

Incrementeel typechecken

Om incrementeel te kunnen typechecken moeten we de resultaten van het controleren van de diverse programma-onderdelen onthouden. We gebruiken daarvoor de abstracte syntax boom. Bij iedere knoop in de boom slaan we het resultaat van de typecontrole van de onderliggende boom op in een zogeheten *attribuut*. We bewaren niet alleen de diverse resultaten maar ook de extra informatie die gebruikt is bij het controleren van een bepaald onderdeel van het programma, zoals de tabel die gebruikt wordt om de statements te controleren.

Op pagina 164 staat zo'n geattribueerde abstracte syntax boom. Rechts van een knoop staat steeds het attribuut waarin het resultaat van een typecontrole kan worden opgeslagen. De attributen waarin de tabel kan worden opgeslagen zijn aan de linkerkant van de knopen getekend.

De uitkomst van een typecontrole van een heel programma wordt bepaald door het resultaat van het controleren van de statements in dat programma. We zeggen dan dat het attribuut bij program *afhangt van* het rechter attribuut bij de onderliggende knoop stms. De pijlen tussen de attributen geven de afhankelijkheden aan. De horizontale pijl van het rechter attribuut van de declaraties naar het linker attribuut van de statements



duidt aan dat de tabel die is samengesteld tijdens het controleren van de declaraties wordt gebruikt bij de controle van de statements.

Stel nu dat iemand $\boxed{\mathsf{Y := X + 1}}$ vervangt door $\boxed{\mathsf{Y := Z - 9}}$. De syntaxchecker vervangt het overeenkomstige gedeelte van de boom. Daarna is de typecheckwaarde in vier attributen ongeldig geworden: namelijk die in de attributen rechts van de knopen +, :=, stms, stms en program.

Wanneer in het gewijzigde programma types gecontroleerd moeten worden kan het controleren van de declaraties achterwege blijven, evenals de controle van read X en if (Y > 10) then write Y else write X. Het nieuwe statement Y := Z − 9 wordt wel gecontroleerd. Het resultaat, fout, wordt opgeslagen in de attributen bij + en :=. Ook de drie andere ongeldig geworden attributen krijgen een nieuwe waarde.

Dit proefschrift

Dit proefschrift beschrijft een techniek voor incrementeel typechecken, waarom dan de titel “Incremental Rewriting”?

Een generator van programmeeromgevingen genereert typecheckers en andere hulpmiddelen uit een specificatie. De implementatie van die hulpmiddelen wordt in hoge mate bepaald door het specificatieformalisme. In het ASF+SDF-systeem is dat een algebraisch formalisme en algebraische specificaties worden bijna altijd geimplementeerd als zogenaamde termherschrijfsystemen. Als bij een programma een typecontrole wordt gedaan zeggen we dat een term wordt herschreven. Om de typecontrole incrementeel te kunnen uitvoeren hebben we dus een techniek nodig voor *incrementeel herschrijven*.

De methode om informatie in attributen van de abstracte syntax boom van een programma op te slaan is niet nieuw. Met name in programmeeromgevingen die beschreven zijn met behulp van een ander specificatieformalisme, attributengrammatica's, wordt dit vaak gebruikt. Ik heb die methode gecombineerd met termherschrijven. Daarbij heb ik dankbaar gebruik gemaakt van een artikel van Courcelle en Franchi-Zannettacci [CFZ82] waarin ze een klasse van algebraische specificaties aangeven die equivalent zijn aan een bepaald soort attributengrammatica. Mijn techniek voor incrementeel herschrijven kan worden toegepast voor alle specificaties in die klasse. De meeste specificaties van typecheckers horen daarbij, maar ook specificaties van bijvoorbeeld vertalers en pretty-printers. In hoofdstuk 2 wordt die klasse van specificaties gedefinieerd en beschrijf ik de algoritmen voor incrementeel herschrijven.

In de hoofdstukken 3 en 4 wordt aangegeven hoe de algoritmen uit hoofdstuk 2 moeten worden aangepast om een wat ruimere klasse van specificaties aan te kunnen.

Aan alle incrementele methoden die gebaseerd zijn op het opslaan van informatie in attributen van een abstracte syntax boom kleeft één groot nadeel. Als een declaratie van een variabele wordt veranderd moeten alle statements opnieuw gecontroleerd worden. Dat is vaak overbodig en duurder dan een niet-incrementele controle. In het programma in het voorbeeld blijkt dat als de declaratie van Z : natural zou worden toegevoegd bijna alle attributen ongeldig worden. In hoofdstuk 5 wordt een oplossing voor dit probleem beschreven.

De diverse technieken zijn geimplementeerd en maken deel uit van het ASF+SDF-systeem. In hoofdstuk 6 wordt de implementatie beschreven. Ik heb voor een aantal programma's gemeten hoeveel sneller een incrementele typecontrole gaat dan een niet-incrementele controle. In hoofdstuk 7 staan de resultaten van deze metingen.

Tenslotte staan in hoofdstuk 8 de conclusies. De methode blijkt bruikbaar en effectief te zijn en is bij een aantal serieuze specificaties met succes gebruikt.