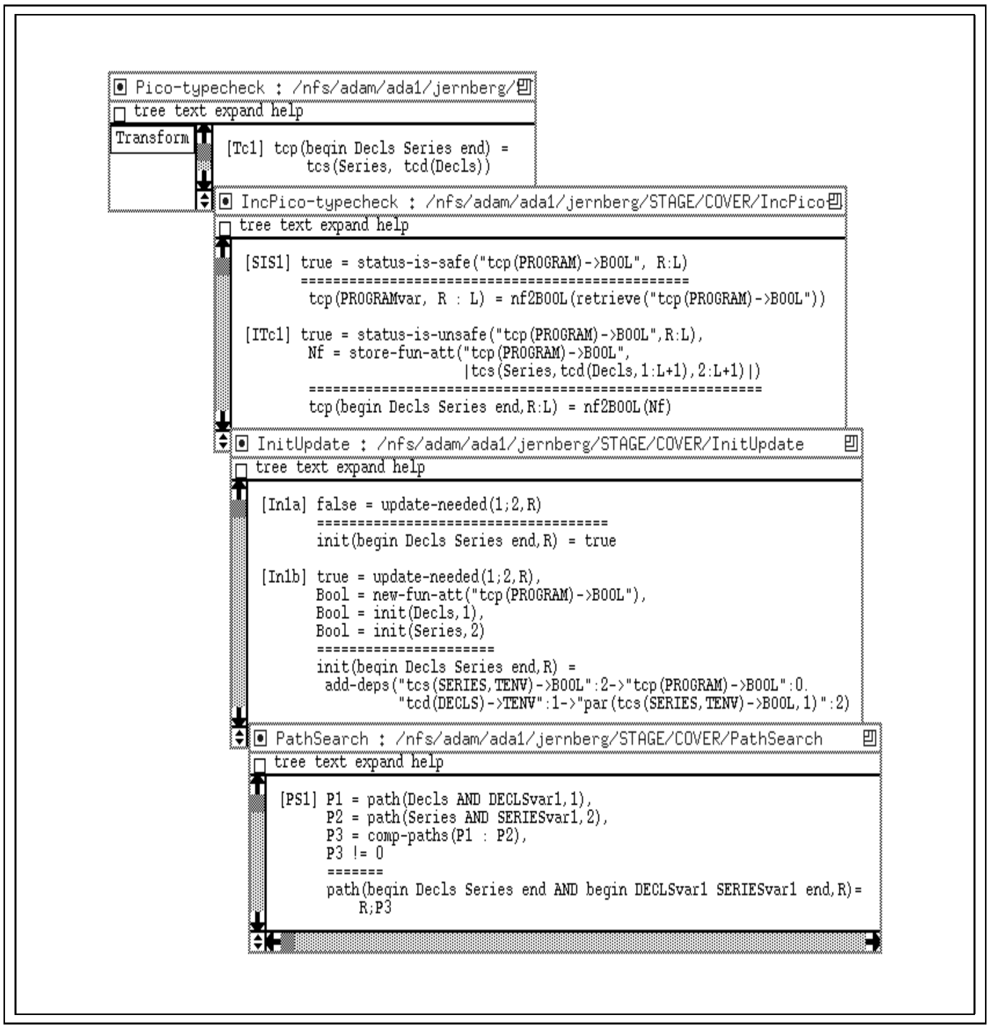
Incremental specifications



Dan Anders Jernberg

1995

Incremental specifications

Dan Anders Jernberg

Doctoraalscriptie Informatica
Afstudeerdocent:
prof. dr. P. Klint (UvA)
Stagebegeleidster:
dr. E. A. van der Meulen (UvA)

Dan Anders Jernberg
Fluitekruidweg 149
1508 AD ZAANDAM
The Netherlands

tel. 075-16 06 42
Ck-nr: 918 81 85
e-mail: jernberg@fwi.uva.nl

Universiteit van Amsterdam 1995

Preface

First of all, I would like to mention Els Penseel at the Joke Smit School in Amsterdam. Thanks to her, I learned enough Dutch to be able to start studying at the University of Amsterdam. My supervisor, Emma van der Meulen, supported me during the hard periods and read various drafts of this thesis, for which I am grateful. She always listened to my problems and had the ability to suggest possible solutions. Mark van den Brand reviewed this thesis thoroughly, for which I am very grateful. Pum Walters and Jasper Kamperman informed me about the new version of hybrid functions, the external functions that are under development. I had a great time and lots of laughs with my fellow students, to mention, Ralph, Fred, Joshua, Mehran, Saskia and Dennis. I want to pay my respect to my grandmother Anna Jernberg and my grandfather Elis Sundin, who both passed away during the last period of this project. Due to lack of time I was not able to attend their funerals; may they rest in peace. Last but not least, my gratitudes to my room mate and good friend Henk Sombroek, for his support, encouragement and good times.

Contents

Chapter 1

Introduction

1.1 Incremental computation

Incremental methods are optimizing techniques for implementation of software. An incremental computation stores the results of subcomputations and reuses the results rather than recomputing them again. It is especially effective when a complex computation is performed over slightly different inputs, or performed over large data structures, to which relatively small modifications are made frequently.

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 containing 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 recomputed. If the computation is incremental, only the value of cells depending on the modified value are recomputed.

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 code generator. 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 with respect to 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 computation again.

In this thesis we are interested in incremental computation in interactive programming environments.

1.2 Prior approach

Our research presented in this thesis is a succession of the research by van der Meulen [Meu94] that was done in the context of the ASF+SDF Meta-environment [Kli93b], a programming environment based on the algebraic formalism ASF+SDF [BHK89b]. An algebraic specification consists of a signature and set of equations. The signature introduces sorts and functions over these sorts and thus defines a set of *terms*. The specification can be made executable by interpreting the equations as a term rewriting system.

In her thesis, van der Meulen presents a method for incremental rewriting. She transferred techniques developed for attribute grammars to algebraic specifications and used attributed trees for storing normal forms for incremental evaluation of terms. 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.

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 value.

This strategy has been implemented as part of the term rewriting engine of the ASF+SDF Meta-environment and 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.

The implementation of incremental rewriting accomplished by van der Meulen is closely connected to one implementation of the term rewriting system, which is currently implemented in LeLisp. The disadvantage of her approach is that if the implementation of the term rewrite engine is changed, the implementation of incremental rewriting has to be adjusted or in the worst case completely reimplemented.

1.3 This thesis

1.3.1 Incremental methods

There are several possibilities to accomplish incremental reduction in a term rewriting system. One method is to add extra code for incremental rewriting to the rewrite engine as described above. Another option is to transform the specifications in such a manner that incremental reduction is guaranteed. To be able to accomplish side-effects in the last approach an "external" storage must be available for storing and retrieving intermediate results.

The implementation of an incremental option in the rewrite engine is for a specific rewrite engine. One can then focus on efficiency issues and the abstract syntax tree that is available can be attributed avoiding the use of a separated storage. If parts of the specification in which incremental functions have been declared are changed, the user will not notice the adjustments made to continue the incremental reduction. If the implementation of the rewrite engine is changed, the extra code for incremental rewriting must be adjusted or in the worst case reimplemented.

When incremental specifications are used only small adaptations in the implementation of the rewrite engine are needed to guarantee incremental rewriting after changes in the implementation of the rewrite engine or when the use of incremental specifications is introduced in a term rewrite system. The transformations from "standard" specifications to incremental specifications can be specified in ASF+SDF and such a specification is easier to understand and maintain than, for instance, C or LeLisp code. An "external" storage is needed to accomplish side-effects when incremental specifications are used. When this storage has once been implemented in a conventional programming language, this implementation can be reused in other systems. A disadvantage of transforming specifications into incremental specifications is that when the original specification is changed, it has to be transformed again.

One limitation of incremental specifications is that rather sophisticated *hybrid* or *external* functions should be supported by the implementation of the term rewrite system. *Hybrid* and *external* functions allow you to define the meaning of a function in your specification by writing a conventional program for it instead of conditional equations. When, during rewriting, this function has to be reduced, this is done by calling the conventional program and not by applying rewrite rules. When a specification is transformed into an incremental specification conditions will be added to the equations defining incremental functions. In these conditions *external* functions will be invoked for checking if a specific value stored in the "external" storage is reliable or not. If a value is reliable, an external function is invoked to retrieve the value from the storage. When the value is not reliable, the term is reduced and the result is stored in the external storage by means of other external functions.

1.3.2 Approach

In this thesis we present a method for incremental reduction with (transformed) incremental specifications written in ASF+SDF so that independent of the implementation of the term rewrite engine, terms over these transformed specifications will be rewritten incrementally.

We have in ASF+SDF specified a storage structure for storing results of evaluations that can be retrieved when re-evaluation is performed. The storage is a table (structure) simulating an *attributed tree*. If we want to apply the function ϕ to the abstract syntax tree T of a text in an editor, the attributed tree we use as storage S should correspond to T. This makes it possible to have a reference to a *current node* in the *attributed tree* at any time during reduction to avoid time consuming search actions. The current node is a node in the attributed tree S corresponding to the node in the abstract syntax tree T that is at that point being evaluated. The specified storage structure can be implemented in any conventional programming language and the only restriction is that the ASF+SDF Meta-environment should support rather sophisticated *hybrid functions* for that programming language.

A user of the ASF+SDF Meta-environment can decide which terms are to be reduced incrementally by attaching an attribute *"incremental"* to the definition of certain functions. Before we can transform specifications into so-called *incremental specifications*, we have to make sure that the *incremental functions* and their rewrite rules form a *primitive recursive scheme* (PRS). A PRS is a subclass of algebraic specifications. We have specified a check in ASF+SDF on specifications to make sure their incremental functions form a primitive recursive scheme.

We call a specification an *incremental specification* if terms over *incremental functions* can be reduced incrementally without an incremental option in the rewrite engine. These incremental specifications are able to invoke functions for storing new values or retrieving old ones. An incremental specification can, for instance, be a specification of a type checker. When a term defined by this incremental specification is to be reduced, it will be reduced incrementally by any term rewrite engine.

Example 1.1 The equation [Tc2] is from a "standard" specification of a type checker, see Appendix B.6. tcs is a function for type-checking a series of statements. tcstat is a function for type checking one statement. Both functions take a type-environment as second argument. Assume that tcs is an incremental function. Equations [Tc2a] and [Tc2b] represents equation [Tc2] after the specification has been transformed into an incremental specification.

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) \ = \ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad \text{[Tc2]}$$

$$\frac{\mathsf{value\text{-}in\text{-}storage\text{-}of}(\texttt{"tcs(SERIES,TENV)->BOOL"}) = \mathit{reliable}}{\begin{array}{l}\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) = \\ \mathsf{retrieve\text{-}value\text{-}of}(\texttt{"tcs(SERIES,TENV)->BOOL"})\end{array}} \qquad \text{[Tc2a]}$$

$$\frac{\begin{array}{r}\mathsf{value\text{-}in\text{-}storage\text{-}of}(\texttt{"tcs(SERIES,TENV)->BOOL"}) = \mathit{unreliable}, \\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) = \mathit{New\text{-}value}\end{array}}{\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) = \mathsf{store\text{-}value\text{-}of}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ \mathit{New\text{-}value})} \qquad \text{[Tc2b]}$$

The equations [Tc2a] and [Tc2b] given above have been extremely simplified but they show the basic principle of incremental specifications. Incremental application of the first equation [Tc2] can only be accomplished when the rewrite engine has an incremental option implemented. In the other two equations, [Tc2a] and [Tc2b], the specification will invoke (external) functions that can approach the storage and influence the rewrite engine to take proper actions by returning suitable normal forms. Equation [Tc2a] applies if the value of a former computation is still valid and the value will be retrieved from the storage and returned as normal form. Equation [Tc2b] applies if the value is unreliable and the normal form will be (re-)computed and stored in the storage. □

An ASF+SDF specification consists of a number of modules linked together by an import mechanism. The whole specification will not be transformed, only the modules in which incremental functions have been declared. We call such a transformed module an incremental module. In addition to the incremental modules, two other modules will be derived from the original specification:

- an *InitUpdate* module that will be used to initialise the storage for new terms that are to be evaluated and to update the storage for terms that have been changed.
- a *SearchPath* module that will provide information about which part of the storage must be updated or re-initialised.

An incremental specification is the combination of the incremental modules, other imported modules, the InitUpdate and the SearchPath modules together with the "external" storage and the external functions.

1.3.3 Results

During this research we have in ASF+SDF specified the following:

- The transformation of a module into an incremental module.
- How a module to initialise and update the storage is derived out of a number of modules.
- How a module to find the part of the storage that corresponds to the part of the abstract syntax tree that must be changed out of a number of modules.
- A check to make sure that a specification is a Primitive recursive scheme with parameters (PRS).
- An "external" storage. This specification also includes functions defined for the different objects in the storage.

The ASF+SDF-specification defining the transformation and the derivations can be compiled to C-code enabling the transformation and the derivation to take place at Unix-level, using the already existing pretty-printer ASFIX2ASDF. It is also possible to combine this C-code with compiled code for the PRS-check yielding a file with error-messages if errors are found during the check, or the transformed and derived modules when the input modules form a PRS.

Even though it should be possible to compile the specification of the storage we use for the incremental evaluation into C-code, we have doubts about the efficiency that would be achieved. Our point of view is that the storage and the functions defined for this structure should be implemented by hand. By doing so, we can really profit from the power of pointer manipulation. We have specified the storage structure in such a manner that it simulates a tree by using tables and we have also simulated pointers by using unique numbers and combinations of these numbers and names for each entrance in the tables.

The reason why we have not yet implemented the storage in a conventional programming language is that the hybrid functions supported by the current implementation of ASF+SDF are too restricted to be useful for our purpose. The use of external C functions will, however, be supported shortly. External C functions are almost the same as hybrid functions and it will be possible to define these functions in the conditions of an equation. The external C functions will be able to accept ASF+SDF-terms as input, whereas the existing hybrid functions are only able to deal with lexical terms.

Before the incremental specifications can be used, there are also minor adjustments needed in the system of which the most important one is to implement a mapping between different editors and their corresponding storages.

1.3.4 Overview

This research has been done in the context of the ASF+SDF Meta-environment. In Chapter 2 an introduction is given to this environment. We will also give the basic definitions of Primitive recursive schemes and at the same time explain why we demand that a specification is a PRS before it can be transformed into an incremental specification.

If an ASF+SDF specification is represented in ASFIX, it is possible to specify operations on the ASFIX specification in ASF+SDF. We will use this option to check whether a specification is a PRS and to define the transformations of the specifications into incremental specifications. Chapter 3 gives an introduction to ASFIX.

We have relaxed some of the conditions a PRS lays on a specification. In Chapter 4 we will explain why and present a list of the remaining conditions together with the method we Use to check these conditions on a specification.

In Chapter 5 we will define the structure we use for storing and retrieving results of an incremental reduction of a term.

In Chapter 6 we will describe the transformations of standard specifications to incremental specifications.

In Chapter 7 we will explain how the storage for incremental reduction can be initialised and updated. We will also describe how we can determine what parts of the storage must be updated or initialised after some parts of the text in the editor has been changed. In this chapter, we will also describe how functions in an incremental specification can invoke (hybrid) functions for storing and retrieving normal forms at reduction time. Interesting is the way the storage can provide information to influence the steps taken by the term rewrite engine. The specification of the propagation algorithms adopted from [Meu94] will also be introduced and elucidated.

In Chapter 8 we conclude with a summary, a discussion of the limitations and suggestions for further research.

The Appendix contains listings of the specifications produced during this research. We will, throughout this thesis, refer to some of these specifications to satisfy readers with the urge to see it all.

Chapter 2

Preliminaries

2.1 Introduction

The research presented in this thesis has been done in the context of the ASF+SDF Meta-environment. In Section 2.2 we will give an introduction to this environment by starting with the formalisms it is based on and continuing with some simple examples. In Section 2.3 we present the toy language Pico and a type checker defined for this language. This type checker will serve as our running example. Before we can start transforming specifications, they have to have certain properties. These properties can be derived from the theory of *Well-presented Primitive recursive schemes with parameters* (PRS). In Section 2.4 we present the basis theorie of PRSs by giving some definitions and discussing why a specification must satisfy certain properties.

2.2 ASF+SDF Meta-environment

The ASF+SDF Meta-environment [Kli93b] is an interactive system for manipulating programs, specifications, or other texts. The formalism that is used in the Meta-environment is ASF+SDF [HHKR92]. ASF+SDF is the result of the marriage of two formalisms ASF (Algebraic Specification Formalism) [BHK89a, BHK89b] and SDF (Syntax Definition Formalism) [HHKR92]. ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. Modules can be imported in other modules and can be parameterized. SDF allows the simultaneous definition of concrete (i.e., lexical and context-free) and abstract syntax.

The main idea of ASF+SDF is to identify the abstract syntax defined by the signature in ASF specifications with the abstract syntax defined implicitly by an SDF specification, thus yielding a standard mapping from text to abstract syntax tree. This gives the possibility to introduce user-defined notation in specifications.

The Meta-environment generates both scanners and parsers from ASF+SDF specifications as well as term rewriting systems, thus allowing execution of ASF+SDF specifications. It also provides syntax-directed editors for the modules in the specification and generates language specific syntax directed editors. The system is able to perform several semantic checks on the specification and supports the testing of the specifications. It can be used to specify arbitrary (programming) languages. This means that both *syntax* and *semantics* of a language can be defined by this system.

2.2.1 ASF+SDF formalism

An ASF+SDF specification consists of a sequence of module declarations. Each module may define syntax rules as well as semantic rules and the notation used in the semantic rules depends on the definition of syntax rules. The entities declared in a module may be visible or invisible to other modules. A module can use other modules from the specification by importing them. As a result, all visible entities of an imported module become available in the importing module. The overall structure of a module is:

imports *Names-of-imported-modules*
  section$*$
priorities
  Priority-relations-between-context-free-rules
equations
  Conditional-equations

A module thus consists of an imports clause followed by a list of zero or more sections (that define sorts, lexical syntax, context-free syntax and variables) optionally followed by a priority declaration and conditional equations. Each section starts with one of the keywords exports or hiddens: exports makes all entities in the section visible to other modules and hiddens makes all entities in the section local to the module. A section has the form:

exports-or-**hiddens**
  sorts *Names-of-sorts*
  lexical syntax
    Rules-of-the-lexical-syntax
  context-free syntax
    Rules-of-the-context-free-syntax
  variables
    Names-of-variables

where each of the subsections for sorts, lexical syntax, context-free syntax and variables is optional.

Example
A simple example specification is given below consisting of the modules Layout, Section 2.2.2 and and Booleans, Section 2.2.3. Module Booleans defines the Boolean values and operators. The syntax of the Boolean language is defined by introducing a sort BOOL, which contains two constants true and false. Furthermore, the standard Booleans functions and (represented as &), or (represented as |) and not are defined.

In order to avoid ambiguities in parsing terms attributes **{left}** or **{right}** can be added to function declarations, defining a function to be left associative or right associative. For instance, the attribute **{left}** in BOOL "&" BOOL → BOOL indicates that the term true & false & true should be parsed as (true & false) & true rather than true & (false & true). Moreover, the specifier can indicate by means of priorities rules how terms should be parsed. For instance, the priority rule in module Booleans states that the operator "&" binds stronger than the operator "|". Hence, the term true | false & true will be parsed as true | (false & true) rather than (true | false) & true. The function with the attribute **{bracket}** is added only for grouping and disambiguation, it is not included in the abstract syntax.

The variable declaration "*Bool*[0-9']$*$ → BOOL" declares an infinite number of variables of sort BOOL. All variables start with the letters *Bool* followed by zero or more ($*$) occurrences of numbers and quotes [0-9']; *Bool2*, *Bool* and *Bool2'* are all variables of the sort BOOL. In the equations we use only one variable.

Module Booleans imports module Layout which consists of three lexical functions for the predefined sort LAYOUT. The first function defines a comment convention by stating that any string starting with two percentage signs ("%%") followed by any number of characters other than a newline (~[\n]), and concluded by a newline ([\n]) is to be considered layout. The second function defines that any string between two percent signs containing any number of characters other than a newline will be regarded as layout as well. The last function defines that spaces (␣), tabs (\t) and newlines (\n) are layout, and thus separate tokens.

In addition to the associativity attributes that can be added to the function declaration in the context-free section of a module, it is also possibly to add the attribute **{incremental}** to the function declaration. By doing this, the user can indicate that the terms over that function should be reduced incrementally.

2.2.2 Layout

```
module Layout
exports
  lexical syntax
    "%%"~[\n]*[\n] → LAYOUT
    "%"~[\n]*"%"   → LAYOUT
    [␣\t\n]        → LAYOUT
```

2.2.3 Booleans

```
module Booleans
imports Layout^A.1
exports
  sorts BOOL
  context-free syntax
    true                → BOOL
    false               → BOOL
    BOOL "|" BOOL       → BOOL  {left}
    BOOL "&" BOOL       → BOOL  {left}
    not "(" BOOL ")"    → BOOL
    "(" BOOL ")"        → BOOL  {bracket}
  variables
    Bool [0-9']* → BOOL
  priorities
    "|" < "&"

equations

  true | Bool   = true     [B1]
  false | Bool  = Bool     [B2]
  true & Bool   = Bool     [B3]
  false & Bool  = false    [B4]
  not(false)    = true     [B5]
  not(true)     = false    [B6]
```

2.3 The toy language Pico

In this section we will have a closer look at the specification of the toy language Pico. We first discuss the module Pico-syntax where the syntax of this simple programming language is defined. We will continue with presenting the module Pico-typecheck. In this module a type checker is presented for the Pico-language. This type checker will serve as our running example throughout this thesis.

2.3.1 Pico-syntax

Below we show the syntax of the language Pico. The module Pico-syntax imports four other modules directly: Identifiers, Integers, Types and Strings. These modules can be found in the Appendix: B.1, A.3, B.3, B.2, and will not be presented here. Pico is a simple programming language. A Pico program has two parts surrounded by the keywords begin and end. In the first part the variables are declared and in the second the statements one wishes to use can be implemented. The declaration part consists of the keyword declare followed by a list of one or more variables (identifiers) and their types, separated by commas. The declaration section is closed with a semicolon. An identifier always starts with a lower case letter that is followed by zero or more lower case letters or numbers. A variable is either of type string or of type natural. The statements section contains a list of one or more statements separated by semicolons. There are three different statements defined for the language Pico: assignment, if-then-else-fi and while-do-od. In all of these three statement expressions occur. An simple expression is an identifier, a natural or a string. There are also composed expressions, such as adding ("+") and concatenating ("||") two expressions or subtracting ("−") one expression from another expression.

module **Pico-syntax**
imports Identifiers$^{B.1}$ Integers$^{A.3}$ Types$^{B.3}$ Strings$^{B.2}$
exports
sorts PROGRAM DECLS ID-TYPE-LIST ID-TYPE SERIES STATEMENT EXP
context-free syntax

begin DECLS SERIES end	$\rightarrow$ PROGRAM	
declare ID-TYPE-LIST ";"	$\rightarrow$ DECLS	
ID-TYPE "," ID-TYPE-LIST	$\rightarrow$ ID-TYPE-LIST	
ID-TYPE	$\rightarrow$ ID-TYPE-LIST	
ID ":" TYPE	$\rightarrow$ ID-TYPE	
STATEMENT ";" SERIES	$\rightarrow$ SERIES	
STATEMENT	$\rightarrow$ SERIES	
ID ":=" EXP	$\rightarrow$ STATEMENT	
if EXP then SERIES else SERIES fi	$\rightarrow$ STATEMENT	
while EXP do SERIES od	$\rightarrow$ STATEMENT	
ID	$\rightarrow$ EXP	
NAT-CON	$\rightarrow$ EXP	
STR-CON	$\rightarrow$ EXP	
EXP "+" EXP	$\rightarrow$ EXP	{**left**}
EXP "−" EXP	$\rightarrow$ EXP	{**left**}
EXP "‖" EXP	$\rightarrow$ EXP	{**left**}
"(" EXP ")"	$\rightarrow$ EXP	{**bracket**}

variables

Decls [*0-9'*]*	$\rightarrow$ DECLS
Id-type-list [*0-9*]*	$\rightarrow$ ID-TYPE-LIST
Id-type [*0-9*]*	$\rightarrow$ ID-TYPE
Series [*0-9'*]*	$\rightarrow$ SERIES
Stat [*0-9'*]*	$\rightarrow$ STATEMENT
Exp [*0-9'*]*	$\rightarrow$ EXP

2.3.2 Pico-typecheck

Below we show the module Pico-typecheck. In this module a type checker for the Pico-language is defined. This module imports three other modules directly. The modules Booleans and Pico-syntax have been discussed already. In module Type-environments a type environment is defined.

To put it very simple, a type environment consists of pairs of identifiers and their types; natural or string. This environment, TENV, is used to check whether all variables that are used in the statements are declared and have the correct type. The module Type-environments can be found in the Appendix B.4. The functions tcp, tcd, tcidlist, tcid, tcs, tcstat and tcexpr are introduced to specify the type checking. All these functions except tcid are marked with the attribute **{incremental}** and terms over these functions are to be reduced incrementally. Of course, we could have declared the function tcid as incremental but we chose not to. The equations defining the rewrite rules for these functions will be explained in the equation section of the module.

module **Pico-typecheck**

imports Booleans$^{A.2}$ Pico-syntax$^{B.5}$ Type-environments$^{B.4}$
exports
context-free syntax

```
tcp(PROGRAM)                     → BOOL  {incremental}
tcd(DECLS)                       → TENV  {incremental}
tcidlist(ID-TYPE-LIST, TENV)     → TENV  {incremental}
tcid(ID-TYPE)                    → PAIR
tcs(SERIES, TENV)                → BOOL  {incremental}
tcstat(STATEMENT, TENV)          → BOOL  {incremental}
tcexpr(EXP, TENV)                → TYPE  {incremental}
```

equations

According to equation [Tc1] below 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 list of pair of variables and their types. The function tcp returns true if all variables in the statements have been declared and have the right type, otherwise it returns false.

tcp(begin *Decls Series* end) = tcs(*Series*, tcd(*Decls*)) [Tc1]

Equation [Tc2] below defines the type checking of the declarations by means of type checking the list of identifiers and their types (the *Id-type-list*) given in an empty type-environment. The function tcd returns a (non-empty) type-environment.

tcd(declare *Id-type-list*;) = tcidlist(*Id-type-list*, empty-tenv) [Tc2]

Equations [Tc3] and [Tc4] below define the type checking of the *Id-type-list*. Equation [Tc3] is applied if the *Id-type-list* consists of only one identifier and its type and this *Id-type* will be type checked by means of the function tcid and added to the type-environment. Equation [Tc4] will be applied if the *Id-type-list* consists of an *Id-type* followed by an *Id-type-list*. In this case the *Id-type-list* will be type checked after the result of type checking the *Id-type* has been added to the type-environment. The function tcidlist yields a type-environment.

tcidlist(*Id-type*, *Tenv*) = tcid(*Id-type*) + *Tenv* [Tc3]

tcidlist(*Id-type*, *Id-type-list*, *Tenv*) = tcidlist(*Id-type-list*, tcid(*Id-type*) + *Tenv*) [Tc4]

Equation [Tc5] below defines the type checking of an identifier and its type and yields the input enclosed with brackets.

tcid(*Id* : *Type*) = (*Id* : *Type*) [Tc5]

Equations [Tc6] and [Tc7] below define the type checking of a *Series* of statements given a type-environment *Tenv*. Equation [Tc6] is applied if the list of statements only contains one statement *Stat*. This *Stat* will then be type checked by the function tcstat in the given type-environment *Tenv*. Equation [Tc7] is applied if the list of statements contains two or more statements and the result of type checking of the first statement will be combined with the type checking of the rest of the statements. The function tcs yields a result of type BOOL; true or false.

$$\mathsf{tcs}(\mathit{Stat},\ \mathit{Tenv})\ =\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv}) \qquad \text{[Tc6]}$$

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv})\ =\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad \text{[Tc7]}$$

Equations [Tc8], [Tc9] and [Tc10] define the type checking of a statement. Equation [Tc8] is applied if the *Statement* is an assignment statement and the type of the identifier will be compared with the result of the type checking of the expression. If the statement is an if-then-else-fi statement equation [Tc9] is applied and yields the results of type checking the the two series of statements *Series1* and *Series2* in the given type-environment *Tenv* if the result of type checking the condition *Exp* yields the type natural. The statement can also be a while-do-od statement and in this case equation [Tc10] is applied where it will be checked that the type checking of the loop condition *Exp* yields the type natural. If this is the case, the function tcstat yields the results of type checking the series of statements in the loop.

$$\mathsf{tcstat}(\mathit{Id} := \mathit{Exp},\ \mathit{Tenv})\ =\ \mathsf{compatible}(\mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id},\ \mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv})) \qquad \text{[Tc8]}$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcs}(\mathit{Series}_1,\ \mathit{Tenv}) = \mathit{Bool}_1,\\ \mathsf{tcs}(\mathit{Series}_2,\ \mathit{Tenv}) = \mathit{Bool}_2\end{array}}{\mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\ \mathit{Tenv}) = \mathit{Bool}_1\ \&\ \mathit{Bool}_2} \qquad \text{[Tc9]}$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) = \mathit{Bool}\end{array}}{\mathsf{tcstat}(\mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od},\ \mathit{Tenv}) = \mathit{Bool}} \qquad \text{[Tc10]}$$

The equations [Tc11], [Tc12] and [Tc13] below define the type checking of the "simple" expressions. If the expression is an identifier, its type will be collected in the type-environment and returned as result. If the expression is a *Nat-con*, one or more numbers, the type natural will be returned and finally if the expression is a *Str-con*, zero or more characters between double quotes, the type string will be returned.

$$\mathsf{tcexpr}(\mathit{Id},\ \mathit{Tenv})\ =\ \mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id} \qquad \text{[Tc11]}$$

$$\mathsf{tcexpr}(\mathit{Nat\text{-}con},\ \mathit{Tenv})\ =\ \mathsf{natural} \qquad \text{[Tc12]}$$

$$\mathsf{tcexpr}(\mathit{Str\text{-}con},\ \mathit{Tenv})\ =\ \mathsf{string} \qquad \text{[Tc13]}$$

The last equations in this module [Tc14], [Tc15] and [Tc16] define the type checking of two epressions containing the binary operators "+", "−" and "||". "+" and "−" represent adding and subtracting two expressions meaning that these expressions should both be of type natural and "||" represents concatenating two string which demands that the expression should be of type string. In all three equations it is checked that the two expression have the right sort given the type-environment *Tenv*.

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}} \qquad \text{[Tc14]}$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 - \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}} \qquad \text{[Tc15]}$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{string},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{string}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1\ \|\ \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{string}} \qquad \text{[Tc16]}$$

2.4 Primitive recursive schemes with parameters

Specifications can only be transformed into incremental specifications if they satisfy certain properties. We start in Section 2.4.1 by giving the definition of a Primitive recursive scheme with parameters (PRS). PRS is a subclass of algebraic specifications. In Section 2.4.2 we continue with the introduction of so-called attributes that we will use for storing information. We proceed by discussing some problems concerning the origins by which the values of the attributes are defined in Section 2.4.3 and we close this section by giving the definition of a well-presented PRS in Section 2.4.4. If a specification satisfies the properties of a well-presented PRS, it can successfully be transformed into an incremental specification.

2.4.1 Intuitive definition PRS

In the definition below we define a primitive recursive scheme as a 5-tuple $\langle G, S, \Phi, E, E_\Phi \rangle$. Intuitively, G is a grammar which consists of a set of free constructors 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 E_Φ specify the behaviour of these Φ-functions. The specification formed by the remaining signature and equations $\langle S, E \rangle$ specifies auxiliary data types for the Φ-functions.

Definition 2.1 (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 $\langle G, S, \Phi, E, E_\Phi \rangle$ 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 $E \cup E_\Phi$. G and S are two disjoint signatures, and Φ is a set of functions. Equations over terms of G do not exist in $E \cup E_\Phi$.

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

(ii) The type of the first argument of each $\phi \in \Phi$ is a sort in G and the types of all other arguments, called the *parameters of* ϕ, and the type of the result sort are sorts of S. Φ_X denotes 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 $p : X_1 \times \cdots \times X_n \to X_0$ in G and each function ϕ in Φ_{X_0} exactly one *defining* equation $e_{\phi,p} \in E_\Phi$ 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.

□

Example 2.1 The Pico-typecheck specification in Section 2.3.2 is a part of a PRS. The abstract tree constructors in G have sorts: PROGRAM, DECLS, ID-TYPE-LIST, SERIES, STATEMENT and EXP. A precise definition of these tree constructors can be found in the module Pico-syntax in Section 2.3.1. Φ is the set of type-check functions that have been declared as *incremental*: {tcp, tcd, tcidlist, tcs, tcstat, tcexpr}. Equations [Tc1]-[Tc4] and [Tc6]-[Tc16] are Φ-defining equations. $\langle S, E \rangle$ is formed by the definition of the function tcid, equation [Tc5] and the specification of the Booleans we discussed in Section 2.2.3 and the Type-environments that can be found in Appendix B.4. □

2.4.2 Attributes

Incremental evaluation of a term means that if a term has already been evaluated we do not evaluate it again. Instead of evaluation we make use of the results of the prior evaluation. In order to reuse results of an evaluation, we have to be able to store these results. For this purpose, we define two sorts of attributes: *function attributes* and *parameter attributes* [1]. Evaluating a term in the context of a rewrite system means reducing a term as far as possible. The result of such a reduction is a *normal form* of the term. When a term $\phi(T)$ has been reduced, we will store the normal form in the function attribute of the function ϕ that is related to T.

A term can have one or more subterms or arguments. We will refer to the first subterm of a ϕ-term as the *constructor-term* and the rest of the subterms as the *parameter-terms* of the ϕ-term. To be able to distinguish between parameter-terms in a ϕ-term, the k-th parameter of a function ϕ is indicated as par(ϕ,k). A parameter term can be a normal form or a subterm that still has to be reduced to a normal form. During incremental reduction of a term ϕ(C, par(ϕ,*1*),...,par(ϕ,k)), with C a constructor-term, we will store the normal forms of the parameter-terms par(ϕ,*1*),...,par(ϕ,k) in parameter attributes.

In an ASF+SDF-specification it is possible to add the attribute **{incremental}** to the declaration of functions. When a function is declared with this attribute, we call it an *incremental function* and refer to functions without this attribute as *non-incremental functions*. The normal forms of a term with an *incremental function* as top operator will be stored in the function attribute and parameter attributes belonging to that term. Normal forms that are results of reducing a term with an *non-incremental functions* top operator will not be stored. To be able to distinguish function attributes that are different but have the same name and are defined for the same constructor term, we have chosen to name function attributes by using the whole declaration of the function, only removing spaces and attached attributes, such as the attribute **{incremental}**. The parameter attributes are named the same way as the parameter terms but we use the name of the function attribute instead of the function name.

Example 2.2 Let us have a look at the type checker defined in Section B.6. The first sort in each function declaration except tcid (this function is not incremental) is the sort of the constructor term of each function, i.e.: PROGRAM, DECLS, ID-TYPE-LIST, SERIES, STATEMENT and EXP. The definition of the syntax of these sorts can be found in the module Pico-syntax, Section B.5. The functions tcp, tcd, tcidlist, tcs, tcstat and tcexpr are all incremental functions and have as function attribute respectively: tcp(PROGRAM) $\rightarrow$ BOOL, tcd(DECLS) $\rightarrow$ TENV, tcidlist(ID-TYPE-LIST,TENV) $\rightarrow$ TENV, tcs(SERIES,TENV) $\rightarrow$ BOOL, tcstat(STATEMENT,TENV) $\rightarrow$ BOOL and tcexpr(EXP,TENV) $\rightarrow$ TYPE. The functions tcidlist, tcs, tcstat and tcexpr all have one parameter term of sort TENV. These function have, respectively, the following parameter attributes: par(tcidlist(ID-TYPE-LIST,TENV)$\rightarrow$TENV,*1*), par(tcs(SERIES,TENV)$\rightarrow$BOOL,*1*), par(tcstat(STATEMENT,TENV)$\rightarrow$BOOL,*1*) and par(tcexpr(EXP,TENV)$\rightarrow$TYPE,*1*). □

The names of the attributes tend to get quite long and in the rest of this chapter we will only use the abbreviated function name (tcp, tcs, etc.) when discussing the attributes and not the whole function declaration.

2.4.3 Attribute definition rules

It is important that each attribute, both function attributes as well as parameter attributes, are uniquely defined in the equations in a specification. The normal form of a ϕ-term is defined by the right-hand side of the defining equation, meaning that the normal form we store in the function attribute corresponding to this ϕ-term is defined by the right-hand side of the defining equation. We will denote B is defined by A with $B \leftarrow A$. Lets have a look at the two following equations in the Pico-typecheck specification:

tcp(begin *Decls Series* end) = tcs(*Series*, tcd(*Decls*)) [Tc1]

[1] In attribute grammars: synthesized attributes and inherited attributes

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) = \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad [\mathrm{Tc7}]$$

From these two equations we can conclude that tcp ← tcs and tcs ← tcstat & tcs.

For a parameter attribute this is more complicated. We have to be sure that parameter terms are uniquely defined as well. A parameter term on the right-hand side of an equation is defined by the definition of the variable or function that it represents. In the right-hand side of equation [Tc1] above, the parameter term par(tcs,*1*) is defined by the term tcd(*Decls*). On the right-hand side of the equation [Tc7] the parameter terms par(tcstat,*1*) and par(tcs,*1*) are both represented by the variable *Tenv*. This variable is defined by par(tcs,*1*) on the left-hand side of the equation. The variable *Tenv* will be instantiated on the left-hand side of the equation with the value of par(tcs,*1*) and transfer this value to the parameters par(tcs,*1*) and par(tcstat,*1*) thus, par(tcstat,*1*) ← par(tcs,*1*) and par(tcs,*1*) ← par(tcs,*1*). If we combine the derivation rules we have found in the equations [Tc1] and [Tc7] we can conclude that par(tcstat,*1*) ← par(tcs,*1*) ← tcd(*Decls*). In Figure 2.1 we show an attributed tree with its attributes and their definition rules.

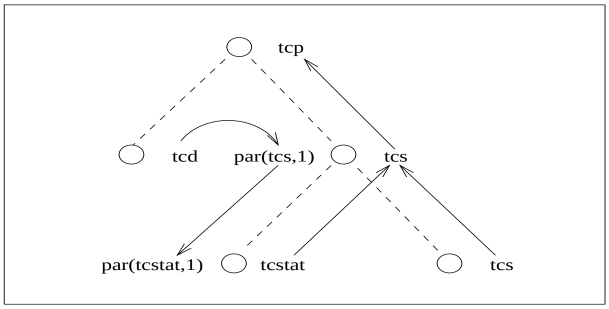


Figure 2.1: Part of an attributed tree with the definition rules shown.

Let us consider another situation:

$$\mathsf{par}(\chi,i) \leftarrow \mathsf{par}(\varphi,j)$$
$$\mathsf{par}(\chi,i) \leftarrow \mathsf{par}(\psi,k)$$

If this is the only information we have, we do not know if $\mathsf{par}(\varphi,j)$ and $\mathsf{par}(\psi,k)$ are both defined by the same origin. If their origins are different, we would not know if the parameter attribute associated with $\mathsf{par}(\chi,i)$ would contain the normal form defined by $\mathsf{par}(\varphi,j)$ or $\mathsf{par}(\psi,k)$ which would result in inconsistency in the stored normal forms. If a specification is a well-presented PRS, however, situations such as described above do not occur and it is guaranteed that all parameter terms can be uniquely defined.

2.4.4 Well-presentedness

Definition 2.2 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

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

$$\phi(q(u_1,\ldots,u_{n'}),z_1,\ldots,z_m) = \tau'$$

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

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

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

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 by $par(\psi, j)$, and the l-th parameter of ξ, $par(\xi, l)$, are represented by the same variable in left-hand sides of defining equations, if and only if for each constructor $p : X_1 \times \cdots \times X_n \rightarrow X_0$ it holds that in right-hand sides of defining equations $\{e_{\phi,p} | \phi \in \Phi_{X_0}\}$ the parameter terms v_j and w_l are identical in all occurences

$$\ldots\psi(x_i, v_1, \ldots, v_k)\ldots \quad \ldots\xi(x_i, w'_1, \ldots, w'_k)\ldots$$

Note that these Φ-terms have the same argument x_i

□

Of course, variable names are not important therefore what counts is whether a PRS is *well-presentable.* That is, it can be transformed into a well-presented PRS by renaming the variables.

Example 2.3 Assume that, for some reason, we want to check that the first statement in a Pico-program uses only the variable x of type natural and we want to be sure that this variable is declared in the declaration section of the program. We have defined this type checker in the module Strange-typechecker below. This module imports the module Pico-typecheck in Section 2.3.2.

imports Pico-typecheck$^{B.6}$
exports
  context-free syntax
    stcp(PROGRAM) → BOOL {**incremental**}
    stcs1(SERIES, TENV) → BOOL {**incremental**}
    stcs2(SERIES, TENV) → BOOL {**incremental**}
equations

stcp(begin *Decls Series* end) = [Stc1]
stcs1(*Series*, (x : natural) + empty-tenv) & stcs2(*Series*, tcd(*Decls*))

stcs1(*Stat*; *Series*, *Tenv*$_1$) = tcstat(*Stat*, *Tenv*$_1$) [Stc2]
stcs2(*Stat*; *Series*, *Tenv*) = tcstat(*Stat*, *Tenv*) [Stc3]

If we look at the equations [Stc2] and [Stc3] and property (v) in Definition 2.2 we notice that the functions stcs1 and stcs2 are defined for the same constructor, represented by: *Stat;Series*. On the right-hand side of both these equations, the function tcstat is defined for the same child of the constructor, represented by the variable *Stat*. If we substitute the variables that represents parameter terms in the equations [Stc2] and [Stc3] we get:

stcs1(*Stat;Series*,par(stcs1,*1*)) = tcstat(*Stat*,par(stcs1,*1*))

stcs2(*Stat;Series*,par(stcs2,*1*)) = tcstat(*Stat*,par(stcs2,*1*))

If we look at the origins of the parameter terms in the equations, it is clear that

par(tcstat,*1*) ← par(stcs1,*1*)
and
par(tcstat,*1*) ← par(stcs2,*1*)

and property (v) in Definition 2.2 does not hold. In Figure 2.2 we show the attributed tree and the definition rules for the strange type checker. If a term is being reduced, par(tcstat,*1*) will contain either the same normal form as par(stcs1,*1*) or the same normal form par(stcs2,*1*) but we can't be sure which. This may cause problems in the incremental computation of stcs2 after that stcs1 has been computed. Since the attribute for tcstat already has a value (based on the value of par(tcstat,*1*), based on the value of par(stcs1,*1*) that is (x : natural) + empty-tenv and not tcd(*Decls*) as it should be) the value is reused rather than computed. □

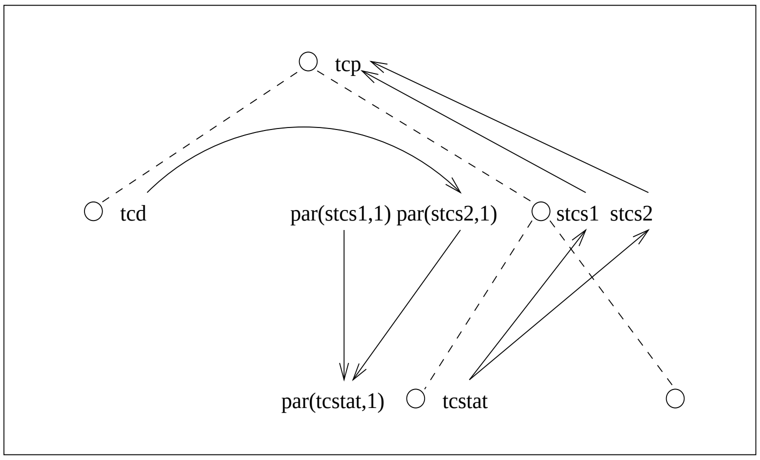


Figure 2.2: Part of the attributed tree for specification that is not a well-presented PRS.

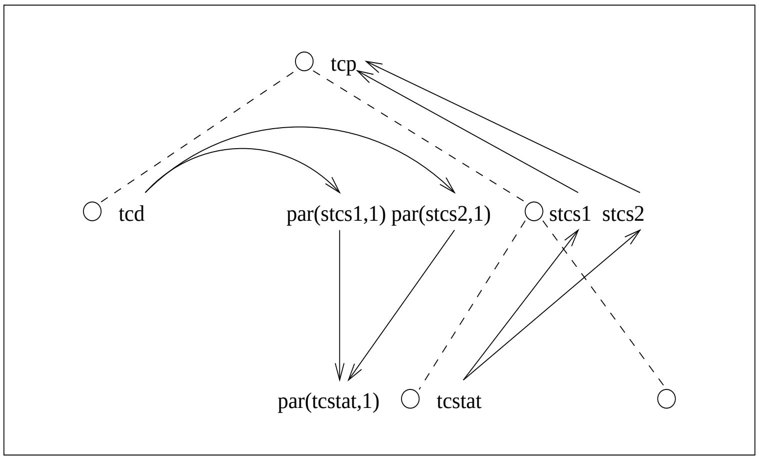


Figure 2.3: Part of the attributed tree for a specification that is a well-presented PRS.

Example 2.4 In this example we change the equation [Stc1] in Example 2.3 into [Stc1b]:

stcp(begin *Decls Series* end) = [Stc1b]
stcs1(*Series*, tcd(*Decls*)) & stcs2(*Series*, tcd(*Decls*))

stcs1(*Stat*; *Series*, $Tenv_1$) = tcstat(*Stat*, $Tenv_1$) [Stc2]
stcs2(*Stat*; *Series*, *Tenv*) = tcstat(*Stat*, *Tenv*) [Stc3]

The rest of the definition of the strange type checker remains the same. The type checker does not really make sense any more, but it will only be used to illustrate property (vi) in the definition of the well-presentedness. The equations [Stc2] and [Stc3] have not been changed, thus we still have that:

$$\mathsf{par}(\mathsf{tcstat},\mathit{1}) \leftarrow \mathsf{par}(\mathsf{stcs1},\mathit{1})$$
$$\text{and}$$
$$\mathsf{par}(\mathsf{tcstat},\mathit{1}) \leftarrow \mathsf{par}(\mathsf{stcs2},\mathit{1})$$

Property (v) in Definition 2.2 still does not hold but if we have a closer look at the right hand side of equation [Stc1b] we can conclude that the parameter terms $\mathsf{par}(\mathsf{stcs1},\mathit{1}) \leftarrow \mathsf{tcd}(\mathit{Decls})$ and $\mathsf{par}(\mathsf{stcs2},\mathit{1}) \leftarrow \mathsf{tcd}(\mathit{Decls})$. Because $\mathsf{par}(\mathsf{stcs1},\mathit{1})$ and $\mathsf{par}(\mathsf{stcs2},\mathit{1})$ are both defined by $\mathsf{tcd}(\mathit{Decls})$ they will always have the same value. If we return to $\mathsf{par}(\mathsf{tcstat},\mathit{1})$ knowing that $\mathsf{par}(\mathsf{stcs1},\mathit{1})$ and $\mathsf{par}(\mathsf{stcs2},\mathit{1})$ always have the same value, we can conclude that:

$$\mathsf{par}(\mathsf{tcstat},\mathit{1}) \leftarrow \mathsf{par}(\mathsf{stcs1},\mathit{1}) \leftarrow \mathsf{tcd}(\mathit{Decls})$$
$$\mathsf{par}(\mathsf{tcstat},\mathit{1}) \leftarrow \mathsf{par}(\mathsf{stcs2},\mathit{1}) \leftarrow \mathsf{tcd}(\mathit{Decls})$$

and the type checker satisfy property (vi) in Definition 2.2 and is well-presented. In this case we regard the two attributes $\mathsf{par}(\mathsf{stcs1},\mathit{1})$ and $\mathsf{par}(\mathsf{stcs2},\mathit{1})$ as one attribute with two names. Figure 2.3 illustrates this situation. □

Chapter 3

AsFix

3.1 Introduction

One of the interesting features of Asf+Sdf is the flexible syntax that can be used in equations: once the context-free syntax of some function has been declared, it can be used freely in equations. As a result, the syntax of the equations part of each module depends on the function declarations.

The syntax of Asf+Sdf is therefore not fixed and for each module a new syntax is constructed consisting of the basic skeleton syntax for conditional equations plus the syntax rules derived from the function declarations. A major disadvantage of this approach is that it is impossible to write "meta-level specifications": Asf+Sdf specifications for manipulating Asf+Sdf specifications.

In [Kli94] Klint presents a fixed exchange format for Asf+Sdf specifications called AsFix. By representing an Asf+Sdf specification in AsFix, a fixed prefix format, the problem with flexible syntax in equations is eliminated.

One advantage of representing a specification in AsFix-notation is that the information in the AsFix specification includes all the information in the original Asf+Sdf specification. A (dis)advantage is that AsFix specifications are intended for automatic processing rather than for human inspection. This means that the readability of the syntax (for humans) is not important. The format, however, is such that writing specifications (in Asf+Sdf) that process the AsFix specification is easy.

In this project we write meta-level specifications. We have specified checks and transformations on Asf+Sdf specifications in Asf+Sdf. In Section 3.2 the data-type ATerm is introduce and some functions defined on this data-type are discussed in Section 3.3. A sort of reference manual for Asf+Sdf specifications in AsFix is given in Section 3.4. Section 3.5 is devoted to global operations on AsFix terms representing (parts of) Asf+Sdf specifications that are needed for this research.

3.2 ATerms: Annotated term structures

A very general data type, ATerm is used to represent AsFix specifications. ATerms are either literal strings or are constructed using function application ("[" and "]"), list constructions (";") or annotations ("/"). Below we give an informal definition of ATerms:

- *Literals*: a literal is an ATerm. Example: `"hello"` and `"43"`.
- *Lists*: the empty list nil as well as expressions of the form $T_1;T_2$ are ATerms. Lists are used to represent sequences of ATerms. Example: `"a";"b";"c"`.
- *Function applications*: Only unary function application is defined and has the form $[T_1\ T_2]$, where T_1 and T_2 are arbitrary ATerms. For instance, `["f" "a"]` is a legal ATerm, but `["f" "a";"b";"c"]` and `[["g" "1";["h" "2"]] "a";"b";"c"]` are also legal ATerms. In a more

conventional notation these functions would be: f(a), f(a,b,c) and g(1,h(2))(a,b,c). At the position where normally a function name would appear, arbitrary ATerms are allowed. In the rest of this thesis we will, however, refer to such an ATerm as the “function symbol”.

- *Annotations*: an ATerm T_1 can be annotated with another ATerm T_2. This is written as T_1 / T_2.

By using only string constants as literals, name clashes in specifications that manipulate ATerms can be avoided; a string constant will appear at the position of the function symbol. At the position where usually the function symbol appears it is possible to write arbitrary ATerms, which enables the possibility to encode arbitrary type information. Both list structures as well as argument lists are needed. By the way the ATerms are defined, it is possible to use operations defined for list structures on argument lists as well. In the next two sections we present the two modules Literals Section 3.2.1 and ATerms Section 3.2.2, in which the Literals and the ATerms are defined.

3.2.1 Literals

Literals are quoted strings constants. They serve the purpose of “constants” in the definition of ATerms.

```
module Literals
imports Layout[A.1]
exports
  sorts Literal
  lexical syntax
    “\”~□              → EscChar
    “\”[01][0-7][0-7]  → EscChar

    ~[\000-\037”\]     → L-Char
    EscChar            → L-Char
    “\”” L-Char* “\””  → Literal
```

3.2.2 ATerms

An ATerm is either a Literal, a list construction (possibly the empty list nil), an application of an annotation.

```
module ATerms
imports Literals[C.1]
exports
  sorts ATerm
  context-free syntax
    Literal                → ATerm
    ATerm “;” ATerm        → ATerm {right}
    nil                    → ATerm
    “[” ATerm ATerm “]”    → ATerm
    ATerm “/” ATerm        → ATerm {left}
    “(” ATerm “)”          → ATerm {bracket}
  variables
    T [0-9′]*        → ATerm
    Ts [0-9′]*       → {ATerm “,”}*
    Ts “+” [0-9′]*   → {ATerm “,”}+
    L [0-9′]*        → Literal
  priorities
```

ATerm "/" ATerm $\to$ ATerm $>$ ATerm ";" ATerm $\to$ ATerm

3.3 Operations on ATerms

It is not difficult to write ASF+SDF specification that perform operations on ATerms. Below the module ATermTools is given. The name of the module and the definition of the function occurs have been adopted from [Kli94]. In the module ATermTools two functions on ATerms are defined: occurs and merge. The function occurs is a Boolean function that determines whether a given term occurs as subterm in another term. The function merge merges two lists of ATerms and at the same time removes all appearances of the empty list nil.

module ATermTools
imports ATerms$^{C.2}$ Booleans$^{A.2}$
exports
  context-free syntax
    occurs(ATerm, ATerm) $\to$ BOOL
    merge(ATerm, ATerm) $\to$ ATerm
equations
The function *occurs* below determines whether a given term appears as subterm in another term.

$$T \neq T_1;\, T_2 \Rightarrow \ \mathsf{occurs}(T,\, T_1;\, T_2) = \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad [\mathsf{Occ1}]$$
$$T \neq [T_1\ T_2] \Rightarrow \ \mathsf{occurs}(T,\, [T_1\ T_2]) = \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad [\mathsf{Occ2}]$$
$$T \neq T_1\ /\ T_2 \Rightarrow \mathsf{occurs}(T,\, T_1\ /\ T_2) = \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad [\mathsf{Occ3}]$$

$$\mathsf{occurs}(T,\, T) = \mathsf{true} \qquad [\mathsf{Occ4}]$$
$$\mathsf{occurs}(T,\, \mathsf{nil}) = \mathsf{false} \qquad [\mathsf{Occ5}]$$
$$\mathsf{occurs}(T_1,\, T_2) = \mathsf{false} \quad \textbf{otherwise} \qquad [\mathsf{Occ}]$$

The function *merge* below takes two list of ATerms as input and produces one list of ATerms. The function *merge* also removes empty (sub) lists (nil) out of the lists. The output produced by the function is a list of ATerms without any nil or the produced list contains only one ATerm, the empty list nil.

$$\mathsf{merge}(T,\, \mathsf{nil}) = T \qquad [\mathsf{Merg1}]$$
$$\mathsf{merge}(\mathsf{nil},\, T) = T \qquad [\mathsf{Merg2}]$$
$$\mathsf{merge}(\mathsf{nil};\, T_1,\, T_2) = \mathsf{merge}(T_1,\, T_2) \qquad [\mathsf{Merg3}]$$
$$\mathsf{merge}(T;\, T_1,\, T_2) = T;\, \mathsf{merge}(T_1,\, T_2) \quad \textbf{when} \ T_1 \neq \mathsf{nil},\ T \neq \mathsf{nil} \qquad [\mathsf{Merg4}]$$
$$\mathsf{merge}(T,\, T_1) = T;\, T_1 \quad \textbf{otherwise} \qquad [\mathsf{Merg}]$$

3.4 ASF+SDF specifications in AsFix

An AsFix specification is simply an ATerm of a certain prescribed form. Since we use ATerms as underlying representation, annotations can be freely inserted wherever permitted by the ATerm syntax. In this section we will explain how an ASF+SDF specification is represented in AsFix. We will start with an entire module and continue with the ATerms representing parts of a module. In the last part of this section we will describe AsFixTerms. AsFixTerms represent ASF+SDF-terms in the right- and left-hand sides of equations and their conditions. This is a rather "ugly" section but useful as reference manual when writing and reading ASF+SDF specifications manipulating ASF+SDF modules, equations and terms in AsFix notation.

A module in AsFix

["Module" L_1;L_2;T_1;T_2;T_3;T_4] This is the ATerm that represents a module. L_1 and L_2 are Literals representing respectively the name of the module and the full module name (directory followed by module name). T_1 and T_3 represent the source text of respectively the syntax part and the equation section of the Asf+Sdf module. T_1 has the form ["SourceSyntax" T] and T_3 looks like ["SourceEquations" T]. In both cases, T is a list of strings. T_2 and T_4 are both function applications where T_2 has the string constant "PrefixSyntax" as function symbol and T_4 has function symbol "PrefixEquations".

["PrefixSyntax" T] This ATerm represents the syntax section in a module in prefix format. T is the empty list nil or a list of sections. These sections are represented by function applications with function symbols the string constants: "Imports", "Exports" and/or "Hiddens".

["PrefixEquations" T] This ATerm represents the equations in a module in prefix format. T is either the empty list nil or a list of ATerms representing the equations. An equation is represented by a function application with function symbol the string constant "CondEquation".

Example 3.1 Here is part of the module Booleans we discussed in Section 2.2.3 in AsFix notation. The SourceSyntax T and the SourceEquations T' are not shown. An example of the PrefixEquations T" will be given when we discuss this section.

```
["Module" "Booleans"; "/nfs/adam/ada1/jernberg/STAGE/IMPL/BIB/Booleans";
 ["SourceSyntax" T];
 ["PrefixSyntax"
  ["Imports" ["Id" "Layout"]];
  ["Exports"
   ["Sorts" ["Sort" "BOOL"]];
   ["ContextFreeSyntax"
    ["CfFunction" ["CfElems" "true"];["Sort" "BOOL"];["Attributes" nil]];
    ["CfFunction" ["CfElems" "false"];["Sort" "BOOL"];["Attributes" nil]];
    ["CfFunction" ["CfElems" ["Sort" "BOOL"];["QLit" "|"];["Sort" "BOOL"]];["Sort" "BOOL"];["Attributes" "left"]];
    ["CfFunction" ["CfElems" ["Sort" "BOOL"];["QLit" "&"];["Sort" "BOOL"]];["Sort" "BOOL"];["Attributes" "left"]];
    ["CfFunction" ["CfElems" "not";["QLit" "("];["Sort" "BOOL"];["QLit" ")"]];["Sort" "BOOL"];["Attributes" nil]];
    ["CfFunction" ["CfElems" ["QLit" "("];["Sort" "BOOL"];["QLit" ")"]];["Sort" "BOOL"];["Attributes" "bracket"]]];
   ["Variables"
    ["Variable" ["LexElems" ["Sort" "Bool"]; ["LexIter" ["CharClass" "[0-9']"]; "*" ]]; ["Sort" "BOOL"]]];
   ["Priorities"
    ["IncrChain"
     ["Abbrev" ["CfElems" ["QLit" "|"]];["Sort" ""];["CfFunction" ["CfElems" "** to be found **"];["Sort" "SORT"];
                                                                                     ["Attributes" nil]]];
     ["Abbrev" ["CfElems" ["QLit" "&"]];["Sort" ""];["CfFunction" ["CfElems" "** to be found **"];["Sort" "SORT"];
                                                                                     ["Attributes" nil]]]
    ]]]];
 ["SourceEquations" T'];
 ["PrefixEquations" T''] ]
```

□

The imports, exports and hiddens sections in AsFix

["Imports" T] This ATerm represents the import section in a module. T is a list of function applications with the string constant "Id" as function symbol.

["Id" L] This ATerm represents the name of an imported module listed in the import section in a module. L is a Literal that contains this module name.

["Exports" T] and ["Hiddens" T] These ATerms represents respectively the export section and hidden section in a module. T is in both cases a list of function applications which can have the following string constants as function symbol: "Sorts", "LexicalSyntax", "ContextFreeSyntax", "Priorities" and/or "Variables".

The sorts section in AsFix

["Sorts" T] This ATerm represents a sorts declaration section in a module. T is a list of function applications of which each should have the string constant "Sort" as function symbol.

["Sort" L] This ATerm represents a sort in a module. L is a Literal that represent the name of the sort.

The context-free syntax section in AsFix

["ContextFreeSyntax" T] This ATerm represents a context-free syntax section in a module. T is a list of function applications representing the context-free function. All the function applications should have the string constant "CfFunction" as function symbol.

["CfFunction" T_1;T_2;T_3] This ATerm represents a context-free function in the context-free syntax section in a module. T_1 is a list of context-free elements, T_2 a result sort and T_3 a list of attributes. T_1, T_2 and T_3 are all function applications and their function symbols are, respectively, the string constants "CfElems", "Sort" and "Attributes".

["CfElems" T] This ATerm represents the context-free elements in a context-free function. T is the empty list nil or a list of Literals and/or function application that can have the following string constants as their function symbols: "Sort", "QLit" and/or "Iter".

["Iter" T_1;T_2;T_3] This ATerm represents an iteration that occurs in a declaration of a context-free function. T_1 is the sort of the iterated element, T_2 is the separator and T_3 is one of the iterators "$\star$" or "+". T_1 is a function application with function symbol the string constant "Sort", T_2 is a function application with function symbol the string constant "QLit".

["Attributes" T] This ATerm represents the attributes attached to a context-free function in the function declarations. T is the empty list nil or a list of Literals representing the attributes.

The variables section in AsFix

["Variables" T] This ATerm represents a variable declaration section in a module. T a list of function applications with function symbol the string constant "Variable".

["Variable" T_1;T_2] This ATerm represents a variable declaration in the variable section of a module. T_1 is a function application with function symbol the string constant "LexElems" and T_2 is a function application with function symbol one of the following string constants: "Sort", "Iter" or "LexIter".

The priorities section in AsFix

["Priorities" T] This ATerm represents the priority section in a module. T is the empty list nil or a list of function applications with as possible function symbols the string constants: "IncrChain","DecrChain" and/or "FunctionList".

["FunctionList" L;T] This ATerm represents a priority chain in the priority section. L is the Literal “<” or “>”. and T is a function application with the string constant "Abbrevs" as function symbol.

["Abbrevs" T] This ATerm represents abbreviations of a rule in priority sections. T is the empty list nil or a list of function applications where each function application has the string constant "Abbrev" as function symbol.

["IncrChain" T] **and** ["DecrChain" T] These ATerms represents increasing or decreasing priority chains in a priority section. T is a list of function applications with the string constants "Abbrev" and/or "FunctionList" as function symbols.

["Abbrev" T_1;T_2;T_3] This ATerm represents an abbreviation in a priority rule. T_1,T_2 and T_3 are all function applications. They have as function symbols, respectively, the string constants: "CfElems", "Sort" and "CfFunction".

The lexical syntax section in AsFix

["LexicalSyntax" T] This ATerm represents a lexical syntax section in a module. T is a list of function applications that all have the string constant "LexicalFunction" as their function symbol.

["LexicalFunction" T_1;T_2] This ATerm represents the declaration of a lexical function in a lexical syntax section. T_1 and T_2 are both function applications. T_1 has as function symbol the string constant "LexElems" and the function symbol of T_2 is the string constant "Sort".

["LexElems" T] This ATerm represents the lexical elements in a lexical function. T is a list of Literals and/or function applications that represent these lexical elements. The function applications can have the following string constants as function symbols: "QLit", "LexIter", "CharClass" and/or "Neg".

["QLit" L] This ATerm represents a quoted Literal. You could say that "QLit" represents the quotes and L represents the Literal between the quotes.

["LexIter" T_1;T_2] This ATerm represent an iteration in a lexical function. T_1 is a list of Literals and/or function applications with the following string constants as possible function symbols: "QLit", "LexIter", "CharClass" and "Neg". T_2 is one of the iterators "$\star$" or "+".

["CharClass" L] This ATerm represents a character class.

["Neg" T] This ATerm represents a negative declaration in a lexical function. T is a list of Literals and/or function application that can have the following string constants as function symbols:"QLit", "LexIter", "CharClass" and/or "Neg". T_2 is one of the iterators "$\star$" or "+".

Example 3.2 The lexical syntax section of the module Layout in Section 2.2.2 in AsFix notation.

```
["LexicalSyntax"
 ["LexicalFunction" ["LexElems" ["QLit" "%%"]; ["LexIter" ["Neg" ["CharClass" "[\\n]"]]; "*" ];
                                ["CharClass" "[\\n]"]]; ["Sort" "LAYOUT"]];
 ["LexicalFunction" ["LexElems" ["QLit" "%"]; ["LexIter" ["Neg" ["CharClass" "[\\n]"]]; "*" ];
                                ["QLit" "%"]]; ["Sort" "LAYOUT"]];
 ["LexicalFunction" ["LexElems" ["CharClass" "[ \\t\\n]"]]; ["Sort" "LAYOUT"] ]]
```

□

The equations in AsFix

["CondEquation" L_1;L_2;T_1;T_2;T_3] This ATerm represents an equation. L_1 is a Literal representing the equation kind: "When" or "Implies". L_2 is a Literal representing the tag of the equation. T_1 is an AsFixTerm representing the left-hand side of the equation and T_2 is also an AsFixTerm but it represents the right-hand side of the equation. T_3 is a function application with as function symbol the string constant "Conditions" and represents the conditions in the equation.

["Conditions" T] This ATerm represents the conditions in an equation. T is the empty list nil or a list of function applications. All these function applications have the string constant "Condition" as their function symbol.

["Condition" L;T_1;T_2] This ATerm represents a condition in an equation. L is a Literal representing the condition kind: "Eq" (equal) and "Neq" (not equal). T_1 and T_2 are both AsFixTerms and represent respectively the right-hand and the left-hand of a condition.

Example 3.3 The equation [B5] not(false) = true and equation [B6] not(true) = false in the module Booleans in ASFIX notation.

```
["CondEquation" "Implies" ; "B5";
 [["CfFunction" ["CfElems" "not";["QLit" "("];["Sort" "BOOL"];["QLit" ")"]];["Sort" "BOOL"];["Attributes" nil]]
  [["CfFunction" ["CfElems" "false"];["Sort" "BOOL"];["Attributes" nil]]  nil]];
 [["CfFunction" ["CfElems" "true"];["Sort" "BOOL"];["Attributes" nil]] nil];
 ["Conditions" nil]] ;
["CondEquation" "Implies" ; "B6";
 [["CfFunction" ["CfElems" "not";["QLit" "("];["Sort" "BOOL"];["QLit" ")"]];["Sort" "BOOL"];["Attributes" nil]]
  [["CfFunction" ["CfElems" "true"];["Sort" "BOOL"];["Attributes" nil]]  nil]];
 [["CfFunction" ["CfElems" "false"];["Sort" "BOOL"];["Attributes" nil]]   nil];
 ["Conditions" nil]]
```

□

AsFixTerms
ASFIXTerms are ATerms that represent ASF+SDF-terms in equations. The empty list nil and the Literals are both ASFIXTerms. The definition of the other ASFIXTerms is as follows.

["Var" T_1;T_2] This ATerm is an ASFIXTerm representing a variable. T_1 represents the variable name and is a Literal or a function application with string constant "QLit". T_2 represents the sort of the variable and is a function application with as function symbol one of the following string constants: "QLit", "Sort", "Iter" or "LexIter".

["Lex" T_1;T_2] This ATerm is an ASFIXTerm representing a lexical string. T_1 represents the lexical string and is either a Literal or a quoted string represented by a function application with string constant "QLit". T_2 represents the sort of the lexical string and is a function application with as function symbol one of the following string constants: "Sort", "Iter" or "LexIter".

[["CfFunction" T_1] T_2] This ATerm is an ASFIXTerm representing an ASF+SDF-term with a context-free function as top operator. The function application with function symbol the string contants "CfFunction" has been described above. T_2 is a list of ASFIXTerms.

[["Iter" T_1] T_2] This ATerm is an ASFIXTerm representing an iterated ASF+SDF-term. The function application with function symbols the string contants "Iter" has been described above. T_2 is a list of ASFIXTerms.

3.5 Operations on ASF+SDF modules in ASFIX

In this section we will briefly discuss some modules we have specified for manipulating ATerms representing ASF+SDF modules and terms. The functions defined in the modules presented below are needed for this project, and are not typical ASFIX operations.

3.5.1 NsATerms: Substituting strings

An ASFIX term contains a lot of strings representing function symbols, variables, sorts etc. In the ASF+SDF system, each character in a string is treated as a separate term. This means that if a string consists of n characters, the string is represented by n separate terms. For efficiency reasons we reduce this number of terms. To reduce the number of terms, we have introduced constants of sort Literal not only for almost all the strings we know will occur in an ATerm, but also for, for instance sorts that we will introduce during the transformations of specifications. A constant is treated as one term. This means that we can reduce the number of terms per string with n-1. Below we show part of the module NsATerms (No string ATerms). The complete module can be found in Appendix D.1. In this module we have defined constants as described above together with two functions: ns-aterms and aterms. The function ns-aterms should be applied before any of the other functions manipulating ATerms is applied. This function substitutes strings for which constants have been defined by these constants. The function aterms should be called when the manipulation is finished, and the constants can be substituted back into the original strings. A typical function call would be: aterms(manipulate(ns-aterms(T))) were T is an ATerm.

module NsATerms
imports ATermTools$^{C.3}$
exports
context-free syntax
ns-aterms(ATerm) → ATerm
aterms(ATerm) → ATerm
scModule → Literal
equations

$$\text{ns-aterms}(T_1;\ T_2) = \text{ns-aterms}(T_1);\ \text{ns-aterms}(T_2) \quad \text{[NsA1]}$$
$$\text{ns-aterms}([T_1\ T_2]) = [\text{ns-aterms}(T_1)\ \text{ns-aterms}(T_2)] \quad \text{[NsA2]}$$
$$\text{ns-aterms}(T_1\ /\ T_2) = \text{ns-aterms}(T_1)\ /\ \text{ns-aterms}(T_2) \quad \text{[NsA3]}$$
$$\text{ns-aterms}(\texttt{"Module"}) = \text{scModule} \quad \text{[NsA4]}$$
$$\text{aterms}(T_1;\ T_2) = \text{aterms}(T_1);\ \text{aterms}(T_2) \quad \text{[AT1]}$$
$$\text{aterms}([T_1\ T_2]) = [\text{aterms}(T_1)\ \text{aterms}(T_2)] \quad \text{[AT2]}$$
$$\text{aterms}(T_1\ /\ T_2) = \text{aterms}(T_1)\ /\ \text{aterms}(T_2) \quad \text{[AT3]}$$
$$\text{aterms}(\text{scModule}) = \texttt{"Module"} \quad \text{[AT4]}$$

3.5.2 Is

The module Is presented below is a small. The functions isList, isInc and isInjection defined in this module are very useful boolean functions.

module Is
imports NsATerms$^{D.1}$
exports
context-free syntax
isList(ATerm) → BOOL
isInc(ATerm) → BOOL
isInjection(ATerm) → BOOL
equations
The function isList below returns true if the input ATerm is a list containing two or more elements, otherwise it returns false. Note that although the empty list nil is a list, the application isList(nil) yields false.

$$\text{isList}(T_1;\ T_2) = \text{true} \quad \text{[IL1]}$$
$$\text{isList}(T) = \text{false} \quad \textbf{otherwise} \quad \text{[IL]}$$

The function isInc below returns true if the Literal scincremental is present as subterm in the input ATerm, otherwise it returns false.

$$\text{isInc}(T) = \text{occurs}(\text{scincremental},\ T) \quad \text{[II1]}$$

The function isInjection below returns true if the input represents an injection, otherwise it returns false.

$$\text{isInjection}([[L\ [L_1\ T];\ T_1;\ T_2]\ [L_2\ L_3;\ T]]) = \text{true} \quad \text{[IInj1]}$$
$$\text{isInjection}(T) = \text{false} \quad \textbf{otherwise} \quad \text{[IInj]}$$

3.5.3 MakeNewATerms: Create new Literals

In the next chapter we will describe the checks we perform on a specification to see whether it is a well-presented PRS. While doing this, information is gathered to produce error messages. These messages will contain the name of the module in which the error was found, which equation it concerns and what term or function declaration is involved. In the module MakeNewATerms we have defined some functions that, given an ATerm representing an ASF+SDF function declaration or an ASF+SDF term, produces a string equal to the original term or function declaration. We show the syntax section of the module MakeNewATerms below and refer to the Appendix D.3 for the complete module.

module MakeNewATerms
imports Integers$^{A.3}$ Is$^{D.2}$
exports
  context-free syntax
    mk-decl(ATerm) → ATerm
    mk-term(ATerm) → ATerm
    mk-decls(ATerm) → ATerm
    mk-terms(ATerm) → ATerm
    cc(ATerm) → ATerm
    ccc(ATerm) → ATerm
    int2lit(INT) → Literal
    lit2int(ATerm) → INT
  variables
    Chars [*0-9'*]∗ → CHAR+
hiddens
  context-free syntax
    mk-term(ATerm, ATerm) → ATerm

Lets say *T* is an ATerm representing the ASF+SDF term tcs(*Series*,tcd(*Decls*)). If the function mk-decl is applied to *T* the result will be the function as it is declared in the syntax section: `"tcs(SERIES,TENV)->BOOL"`. If the function mk-term is applied to the same *T* the result will be: `"tcs(Series,tcd(Decls))"`. The function mk-decls and mk-terms produces a list of function declarations and terms respectively, if their input is a list of ATerms representing ASF+SDF-terms.

The function cc concatenates a list of Literals into one Literal and the function ccc does the same as cc but puts commas between the Literals that are concatenated. The function int2lit converts an integer into a Literal and the function lit2int converts a Literal representing an integer into an integer. These two convert functions enable us to increase or decrease, for instance, variable indices in ATerms.

3.5.4 ComplexPrsProjection: Gather information

When a specification is checked on the PRS properties or is transformed, we are often interested only in specific parts of the specification. Therefore, we have defined some functions that select these specific parts. These functions are defined in the Module ComplexPrsProjections of which the syntax section is shown below. For the equations defining these functions, we refer to the Appendix D.5. The imported module SimplePrsProjections will be discussed in Section 4.2.

module ComplexPrsProjections
imports SimplePrsProjections$^{D.4}$
exports
  context-free syntax
    inc-operators-in-syntax({ATerm ","}∗) → ATerm
    inc-equations({ATerm ","}∗) → ATerm

non-inc-equations({ATerm ","}*) → ATerm
operators-in-all-equations({ATerm ","}*) → ATerm
vars(ATerm) → ATerm
inc-operators(ATerm) → ATerm
operators-in-eqs(ATerm) → ATerm
inc-terms(ATerm) → ATerm
inc-constructors(ATerm) → ATerm
cf-syntax(ATerm) → ATerm

hiddens
context-free syntax
inc-eqs(ATerm) → ATerm
non-inc-eqs(ATerm) → ATerm

The first four functions declared in the module above expect as input a list of ATerms representing ASF+SDF modules. The information is selected per module and a list of annotated ATerms T/L is returned, where T is the information and L is the name of the module in which the information was found. The function inc-operators-in-syntax selects all function declarations of functions that have been declared as "incremental" in the syntax sections of a specification. The function inc-equations selects all equations having a top operator on its left-hand side, that has been declared as incremental. All equations that have a top operator on the left-hand side that has not been declared as incremental, are selected by the function non-inc-equations. The function operators-in-all-equations selects the top operators on the left-hand side of all equations in the input.

The other functions declared in the module do not annotate their output. The function vars selects all ATerms representing variables in its input. All operators that have been declared as incremental are selected by the function inc-operators and the function operators-in-eqs selects all top operators inc-terms selects all ATerms representing ASF+SDF terms that have an incremental top operator. The function inc-constructors selects the first argument of all terms whose top operator has been declared as incremental. Because of the flexibility of ASF+SDF, it is possible to write the context-free syntax section any where in the "hiddens" and/or "exports" sections of an module. The function cf-syntax searches the syntax section of a module and returns the function declarations that have been declared in all the context-free syntax sections that are present in a module.

The "hidden" functions inc-eqs and non-inc-eqs selects, respectively, the incremental and the non-incremental equations in ATerm representing a module.

3.5.5 Reconstruction of equations

The flexibility of ASF+SDF allows users to introduce variables in the conditions. This means that if we leave these new uninstantiated variables in the equations we have to take these variables into account when a specification is PRS-checked or transformed. In the module Restruct-eqs of which a part is given below, we have defined a function that changes the structure of an equation in such a way that these variables are removed. The complete module can be found in Appendix D.6.

module Restruct-eqs
imports ComplexPrsProjections$^{D.5}$
exports
context-free syntax
restruct-eqs(ATerm) → ATerm

Another effect of the function restruct-eqs is that the constructors in the conditions and the right-hand side of the equation are annotated with the rank of the child the constructors represent. Variables in the conditions and the right-hand side of the equation, that are defined by the parameters in the left-hand side of the equation are also annotated. We annotate these variables with the par(ϕ,j) notation where ϕ is the top operator of the equation and j indicates which parameter term is meant. When these annotations have been added to the constructors and the variables, we can easier derive definitions rules in the equations as described in Section 2.4.3. Another advantage is that the annotations contain information needed in various parts of the transformations and the PRS-check and it is not necessary to seek it every time it is needed. The actions of the function restruct-eqs are illustrated in the example below.

Example 3.4 Restructing an equation.
Original equation:

$$\frac{\begin{array}{r@{\;}l} \mathit{Exp}_1 &= \mathit{Exp}, \\ \mathit{Tenv} &= \mathit{Tenv}_1, \\ \mathsf{natural} &= \mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}), \\ \mathit{Bool}_1 &= \mathsf{tcs}(\mathit{Series}_2,\ \mathit{Tenv}_1), \\ \mathit{Bool}_2 &= \mathsf{tcs}(\mathit{Series}_1,\ \mathit{Tenv}) \end{array}}{\begin{array}{l} \mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\ \mathit{Tenv}) = \\ \mathit{Bool}_1\ \&\ \mathit{Bool}_2 \end{array}} \quad [\mathsf{Tcx}]$$

The equation after the introduced variables have been substituted:

$$\frac{\mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv})}{\begin{array}{l} \mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\ \mathit{Tenv}) = \\ \mathsf{tcs}(\mathit{Series}_2,\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series}_1,\ \mathit{Tenv}) \end{array}} \quad [\mathsf{Tcx}]$$

The equation after (sub) constructors and parameters have been annotated:

$$\frac{\mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}\ /\ 1,\ \mathit{Tenv}\ /\ \mathsf{par}(\mathsf{tcstat},1))}{\begin{array}{l} \mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\ \mathit{Tenv}) = \\ \mathsf{tcs}(\mathit{Series}_2\ /\ 2,\ \mathit{Tenv}\ /\ \mathsf{par}(\mathsf{tcstat},1))\ \&\ \mathsf{tcs}(\mathit{Series}_1\ /\ 1,\ \mathit{Tenv}\ /\ \mathsf{par}(\mathsf{tcstat},1)) \end{array}} \quad [\mathsf{Tcx}]$$

□

Chapter 4

PRS check on a specification

4.1 Introduction

Before we transform a specification into an incremental specification, we must be sure that this is possible. In Section 2.4 we have given the definition of a Well-presented Primitive Recursive Scheme with parameters. In this chapter, a method is presented for checking that an ASF+SDF specification in ASFIX notation is a Well-presented PRS. In Section 4.2 we start by repeating the terminology concerning a PRS to facilitate the discussion in the rest of this chapter. We will also present some projection functions on ASF+SDF terms in ASFIX notation. The definition of a PRS is very strict. Some of the constraints the PRS definition imposes on a specification are not necessary for our method of incremental rewriting. We have relaxed these requirements and in Section 4.3 we discuss why this is possible and present the necessary requirements. In Section 4.4 we present the global check followed by the specific checks that have to be performed on the equations with non-incremental top operators on the left-hand side of the equation in Section 4.5. In Section 4.6 we continue with the presentation of checks for the equations with incremental top operator on the left-hand side. In Section 4.7 we describe how the well-presentedness of a specification is checked. Simultaneous with the checks information is gathered for error messages. In Section 4.8 these messages are discussed briefly.

4.2 Terminology

The name of a function will be referred to as the *operator*. We call the first argument in an incremental function the *constructor*. This is not really correct because it first has to be checked that this first argument really is a constructor, but it will facilitate the rest of the discussion. Our constructors should be *abstract tree constructors* of the form $p : X_1 \times \cdots \times X_n \to X_0$. We will refer to $X_1 \cdots X_n$ as the *children* of the constructor. To be able to distinguish these children, we give them a *rank*: the child corresponding to X_1 has rank 1, the child corresponding to X_2 has rank 2 and so forth. All arguments in a function except the first, the constructor, will be referred to as the *parameters* of the function. We refer to the j'th parameter of a function with operator ϕ as $\mathsf{par}(\phi, j)$.

In the module SimplePrsProjections, Appendix D.4, we have specified the functions operator, constructor and parameters. These functions selects, respectively, the operator, the constructor and the parameters in an ATerm representing an ASF+SDF term. If these functions are applied to an ATerm representing an equation, the operator, constructor or the parameters of the left-hand side of the equation is returned. We have chosen this option to facilitate the rest of the specification. All three functions have the default return value nil. The equations defining these functions are shown below.

$$
\begin{array}{llll}
\mathsf{operator}([[\mathsf{scCfFunction}\ T]\ T_1]) & = [\mathsf{scCfFunction}\ T] & & [\mathsf{Op1}] \\
\mathsf{operator}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) & = \mathsf{operator}(T_1) & & [\mathsf{Op2}] \\
\mathsf{operator}(T) & = \mathsf{nil} & \textbf{otherwise} & [\mathsf{Op}]
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{constructor}([[\mathsf{scCfFunction}\ T]\ T_1;\ T_2]) & = T_1 & [\mathsf{Con1}] \\
\mathsf{constructor}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) & = \mathsf{constructor}(T_1) & [\mathsf{Con2}]
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{constructor}([[\mathsf{scCfFunction}\ T]\ T_1]) & = T_1 & & [\mathsf{Con3}] \\
 & \textbf{when} & & \\
 & \quad \mathsf{false} = \mathsf{isList}(T_1) & & \\
\mathsf{constructor}(T) & = \mathsf{nil} & \textbf{otherwise} & [\mathsf{Con}]
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{parameters}([[\mathsf{scCfFunction}\ T]\ T_1;\ T_2]) & = T_2 & & [\mathsf{Par1}] \\
\mathsf{parameters}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) & = \mathsf{parameters}(T_1) & & [\mathsf{Par2}] \\
\mathsf{parameters}(T) & = \mathsf{nil} & \textbf{otherwise} & [\mathsf{Par}]
\end{array}
$$

According to Definition 2.1, (i), a PRS can be described as a 5-tuple $\langle G, S, \Phi, E, E_\Phi \rangle$. We will use this notion to gather the information we need to check that a specification is a PRS. If a function has been declared as incremental in the syntax section of a specification we call it an *incremental function*. Functions that have not been declared as incremental are called *non-incremental functions*. We refer to Φ as the set of all incremental functions. E_ϕ is formed by all equations that have an incremental operator as top operator on its left-hand side. We refer to these equations as *incremental equations*. Equations that do not have an incremental operator as top operator on its left-hand side form the set E and we refer to these equations as *non-incremental equations*.

4.3 Properties checked

In this section we will discuss the properties we check in a specification. We begin with the properties in Definition 2.1. We have to check that there are no equations defined for the constructors of the incremental functions in the specification. It would be fatal for our method of incremental reduction if these constructors could reduce to another term.

We will not check that there is exactly one defining equation defined for each incremental function. The equations in the rewrite system implemented in the ASF+SDF Meta-environment are *conditional equations* and it is possible to define multiple equations defining the semantics of the incremental functions.

It must be checked that the left-hand side of each equation defining the incremental functions are left-linear, i.e. the names of the variables on the left-hand side of the incremental equations are all different.

Property (iii) (b) states that the equations must be strictly decreasing. We have relaxed this property a bit. We allow two situations for the constructors in incremental functions in the conditions and the right-hand of the incremental equations:

- the constructor is a child of the constructor of the left-hand side of the equation.
- the constructor is equal to the constructor of the left-hand side of the equation.

The second situation describes an injection. If we are dealing with an injection, we will give the constructor representing this "child" rank 0.

Example 4.1 The following two equations are part of the Pico-typecheck specification we discussed in Section 2.3.2.

$$\frac{\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{natural},\quad \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}}{\mathsf{tcexpr}(\mathit{Exp}_1\ +\ \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}} \qquad \text{[Tc14]}$$

$$\mathsf{tcs}(\mathit{Stat},\ \mathit{Tenv})\ =\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv}) \qquad \text{[Tc6]}$$

In equation $[Tc14]$ the child of the constructor represented by the variable Exp_1 has rank 1 and Exp_2 has rank 2. In equation $[Tc6]$ an injection is defined thus the child of the constructor is the constructor that is represented by the variable *Stat*. In the right-hand side of the equation, the constructor has rank 0. □

We will now continue with describing the properties of Definition 2.2 we check. Property (iv) states that for any two equations concerning the same incremental function, the names of the variables must be the same, otherwise the specification is not well-presented. In our approach, this is not necessary. We will make this property hold by renaming the variables.

It is essential to check property (v). If, in the conditions or the right-hand side of distinct incremental equations, the same incremental function with the same constructor is applied, the parameters must be uniquely defined. The names of the variables representing these parameters are not essential as stated in property (iv) because we will rename them. If a parameter in an incremental function is not equally defined, we will turn to property (*vi*) to check if the ultimate origins defining these parameters are equal.

There are some additional checks we have to perform on the specification. We are restricted by the size of an ATerm representing an ASF+SDF module. These ATerms are enormous with commonly shortage of memory during runtime as result. If a user wants to transform a specification into an incremental specification, the user will probably transform some of the modules in the specification but not all. We have to check that all the incremental functions that are applied in the to transformed modules are defined in these modules. Otherwise a function call would be transformed but not the definition of the function.

Another check we have to perform is to make sure that an incremental function is not present in a non-incremental equation. The method for incremental rewriting presented in this thesis does not (yet) cover lists, thus a check will be performed to see if constructors are defined by list operators. Incremental functions may not be defined by default equations. We postpone the explanation of this requirement until Section 7.4.7.

4.4 Global check on a specification

In the module PRS-check we have defined the global check we perform on a specification to see if the incremental functions defined in it and their defining equations form a well-presented PRS. In the module PRS-check, we have defined one function: prs-check. The module PRS-check is shown below.

module PRS-check
imports Check-non-inc-eqs$^{E.3}$ Check-inc-eqs$^{E.4}$ Check-WP$^{E.5}$
exports
  context-free syntax

prs-check({ATerm ","}*, {ATerm ","}*) $\rightarrow$ ATerm

equations

The function *prs-check* below expects as input two arguments: the first argument, *Ts+* is a list of ATerms representing the modules that should be checked, and eventually transformed, and the second argument, *Ts* contains a list of ATerms that represents all the modules that are imported by the modules in the first argument. Equation [PC2] applies when the list of ATerm representing the modules is empty.

$$\frac{\begin{array}{l} T_1 = \text{inc-operators-in-syntax}(Ts^+), \\ T_2 = \text{inc-equations}(Ts^+), \\ T_3 = \text{non-inc-equations}(Ts^+), \\ T_4 = \text{operators-in-all-equations}(Ts^+,\ Ts), \\ T_5 = \text{restruct-eqs}(T_2), \\ T_6 = \text{check-non-inc-eqs}(T_3), \\ T_7 = \text{check-inc-eqs}(T_1,\ T_5,\ T_4), \\ T_8 = \text{check-wp}(T_5), \\ T_9 = \text{merge}(T_6,\ \text{merge}(T_7,\ T_8)) \end{array}}{\text{prs-check}(Ts^+,\ Ts) = \text{message}(T_9)} \quad \text{[PC1]}$$

$$\text{prs-check}(,\ Ts) \ = \ \text{message}(\text{nil}) \quad \text{[PC2]}$$

In the first four conditions information needed to perform the checks is gathered. The functions inc-operators-in-syntax, inc-equations, non-inc-equations and operators-in-all-equations are defined in the module ComplexPrsProjections that can be found in Appendix D.5. The function inc-operators-in-syntax returns a list of ATerms. Each ATerm has the form T/L where T is a list of ATerms representing the function declaration of all functions that have been declared as incremental found in L, that represents the name of a module. The function inc-equations also returns a list of ATerms T/L but in this case T represents all incremental equations found in module L. The function non-inc-equations return a list of ATerms similar to the list returned by the function inc-equations but in this case T is a list of all non-incremental equations. The function operators-in-all-equations also returns a list of ATerms annotated with module names. In this case, the ATerms are lists of annotated ATerms where the operator of the left-hand side of an equation is annotated with the tag of that particular equation. These operators will be used to check that there are no equations defined for the constructors of the incremental functions.

The reason why we annotate the gathered information in this manner is that simultaneously with the checks we perform on the specification, error messages will be produced if parts of the specification do not meet with the requirements demanded. The module names and the equation tags will provide us with information which will enable us to inform in which module the error was found and which equation it concerns.

Before the actually check functions are applied, the incremental equations are restructed by the function restruct-eqs as described in Section 3.5.5. The equations defining this function can be found in module Restruct-eqs in Appendix D.6. The last three conditions in the equation defining the function prs-check will be explained in the next three sections. The function message is applied to the results of these three function applications: check-non-inc-eqs, check-inc-eqs and check-wp (well-presentedness). Depending on the input the function message returns the message that the specification can be transformed (the specification is a well-presented PRS) or an error message.

4.5 Checking the non-incremental equations

We have to be sure that no incremental function is applied in an equation with a top operator on the left-hand side of the equation that has not been declared as incremental. We have specificed this check in the module Check-non-inc-eqs shown below.

module **Check-non-inc-eqs**

imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    check-non-inc-eqs(ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    check-modules(ATerm) $\rightarrow$ ATerm
    check-equations(ATerm) $\rightarrow$ ATerm
equations

The function check-non-inc-eqs below expects as input a list of annotated ATerms T/L. T contains the non-incremental equations the module L.

check-non-inc-eqs(T) = check-modules(T) [Cnie]

The function check-modules below checks the non-incremental equations per module. Given the result of checking the equations, the function mod-msg is applied. If the result of checking the equations is nil, the function mod-msg returns nil, otherwise an error message is returned. This message starts with the name of the module in which the errors where found, followed by the information returned by the function check-equations. The name of the module is represented by the L in the equation [$Cmod2$] below.

check-modules(T_1; T_2) = merge(check-modules(T_1), check-modules(T_2)) [Cmod1]
check-modules(T / L) = mod-msg(check-equations(T) / L) [Cmod2]

The function check-equations defined below checks the non-incremental equations individually. This is done by first applying the function inc-terms. This function returns all terms in the equations that have been declared as incremental or nil if no incremental terms where found. The function mk-terms returns strings that look like the terms in ASF+SDF-notation. If no errors are found in an equation the function check-equations returns nil, otherwise it returns an error message per equation indicating which terms are concerned. L_2 in equation [$Ceq2$] below represents the tag of the equation.

check-equations(T_1; T_2) = merge(check-equations(T_1), check-equations(T_2)) [Ceq1]

check-equations([scCondEquation L_1; L_2; T]) = [Ceq2]
eq-msg(non-inc-eq-msg(mk-terms(inc-terms(T)) / L_2))

check-equations(T) = nil **otherwise** [Ceq]

4.6 Checking the incremental equations

In this section we describe the checks we perform on the incremental equations of a specification. As in the previous section, where we described the check of the non-incremental equations, we try to give error messages some structure. The messages will be given per module and per equation. This is why the check might have a kind of arbitrary sequence.

module **Check-inc-eqs**

imports Declared$^{E.4.1}$ Different-vars$^{E.4.2}$ Constructors$^{E.4.3}$ No-constructor-eqs$^{E.4.4}$
exports
  context-free syntax
    check-inc-eqs(ATerm, ATerm, ATerm) $\rightarrow$ ATerm

hiddens

context-free syntax

check-modules(ATerm, ATerm, ATerm) $\to$ ATerm
check-equations(ATerm, ATerm, ATerm) $\to$ ATerm

no-default(ATerm, ATerm) $\to$ ATerm
no-list(ATerm) $\to$ ATerm

variables

Chars [*0-9*]$*$ $\to$ CHAR$*$

equations

The function check-inc-eqs defined below checks all the incremental equations. In equation [$Cie1$], T_1 contains the declarations of the incremental functions that have been found in the syntax sections, T_2 the incremental equations we want to check and T_3 contains the top operators of all equations that are in the modules we are checking and in the modules imported by these modules.

$$\text{check-inc-eqs}(T_1,\ T_2,\ T_3)\ =\ \text{check-modules}(T_1,\ T_2,\ T_3) \qquad \text{[Cie1]}$$

The function check-modules defined below invokes the check on the incremental equations per module. This enables us to report errors per module. If no errors are found in a module nil is returned, otherwise a message is returned with the name of the module followed by a list of errors that where found in that module.

$$\text{check-modules}(T_1,\ T_2;\ T_2{}',\ T_3) = \qquad \text{[Cmod1]}$$
$$\text{merge}(\text{check-modules}(T_1,\ T_2,\ T_3), \text{check-modules}(T_1,\ T_2{}',\ T_3))$$

$$\text{check-modules}(T_1,\ T_2\ /\ L,\ T_3) = \qquad \text{[Cmod2]}$$
$$\text{mod-msg}(\text{check-equations}(T_1,\ T_2,\ T_3)\ /\ L)$$

The function check-equations checks the incremental equations individually, enabling us to report errors found per equation. An incremental equation must satisfy six requirements. The first is that all incremental terms in the equation have been declared in the modules we are checking. This check is performed when the function declared is applied. The next check is to make sure that the equation is left-linear: all variables in the left hand side of the equation have different names. This is done by the function different-vars. The third property we check is that the constructors of the incremental terms in the conditions and on the right-hand side of an incremental equation are either children or equal to the constructor on the left-hand side of the equation. This check is performed by applying the function constructors. The next check assures us that there are no equations defined for the constructor on the left-hand side of this equation. This check is performed by the function no-constructor-eqs. We do not allow equations defining incremental functions to be default equations. This check is performed when the function no-default is applied. The method for incremental rewriting with incremental specifications does not (yet) cover constructors that are defined as list functions. The last check returns a warning if this is the case.

$$\text{check-equations}(T_1,\ T_2;\ T_2{}',\ T_3) = \qquad \text{[Ceq1]}$$
$$\text{merge}(\text{check-equations}(T_1,\ T_2,\ T_3), \text{check-equations}(T_1,\ T_2{}',\ T_3))$$

$$\frac{\begin{array}{l} T_4 = \text{declared}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T), \\ T_5 = \text{different-vars}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]), \\ T_6 = \text{constructors}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]), \\ T_7 = \text{no-constructor-eqs}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T'), \\ T_8 = \text{no-default}(L_2,\ \texttt{"default"}), \\ T_9 = \text{no-list}(\text{constructor}(T_1)), \\ T_{10} = \text{merge}(\text{merge}(T_4,\ T_5), \text{merge}(\text{merge}(T_6,\ T_7), \text{merge}(T_8,\ T_9))) \end{array}}{\begin{array}{l} \text{check-equations}(T, [\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T') = \\ \text{eq-msg}(T_{10}\ /\ L_2) \end{array}} \qquad \text{[Ceq2]}$$

$$\text{check-equations}(T_1,\ T_2,\ T_3)\ =\ \text{nil}\quad \textbf{otherwise} \qquad \text{[Ceq]}$$

The function no-default defined below returns nil if the tag, the first argument is not a default equation tag, otherwise an error message.

$$\text{no-default}(\text{literal}(\texttt{"""}\ \mathit{Chars}\ \mathit{Chars}_1\ \texttt{"""}),\ \text{literal}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""})) = \text{no-default-msg}\ (\ ) \qquad \text{[Ndef1]}$$

$$\text{no-default}(\text{literal}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""}),\ \text{literal}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""})) = \text{no-default-msg}\ (\ ) \qquad \text{[Ndef2]}$$

$$\text{no-default}(T,\ T')\ =\ \text{nil}\quad \textbf{otherwise} \qquad \text{[Ndef]}$$

The function no-list defined below returns nil if the constructor, the input, is not a list construction, otherwise it returns an error message.

$$\text{no-list}(T)\ =\ \text{no-list-msg}\ (\ )\ \ \textbf{when}\ \ \text{true} = \text{occurs}(\text{scIter},\ T) \qquad \text{[Nlist1]}$$

$$\text{no-list}(T)\ =\ \text{nil}\quad \textbf{otherwise} \qquad \text{[Nlist]}$$

4.6.1 All incremental functions declared

It is important that all incremental functions occurring in the modules we want to transform into incremental modules have been declared in these very modules. If, for instance, an equation defining an incremental function is declared in one of the modules we do not transform, the declaration of the function remains unchanged, but its defining equation is transformed. In the module Declared below we have specified this check.

module Declared
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    declared(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    check-operators(ATerm, ATerm) → ATerm
equations

The function declared defined below checks that all incremental functions occuring in an equation have been declared in the modules that are checked. T contains the declarations of all these functions. All incremental terms in the equation are selected by means of function inc-terms and the function check-operators is applied. The function check-operators returns all terms which top operators are not present in T. If all these top operators are present they have all been declared and the function check-operators returns nil.

$$\text{declared}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T) = \text{declared-msg}(\text{mk-terms}(\text{check-operators}(\text{inc-terms}(T_1;\ T_2;\ T_3),\ T))) \qquad \text{[D]}$$

$$\text{check-operators}(T;\ T',\ T_1) = \text{merge}(\text{check-operators}(T,\ T_1),\ \text{check-operators}(T',\ T_1)) \qquad \text{[Co1]}$$

$$\frac{\text{false} = \text{occurs}([\text{scCfFunction}\ T],\ T_1)}{\text{check-operators}([[\text{scCfFunction}\ T]\ T'],\ T_1) = [[\text{scCfFunction}\ T]\ T']} \qquad \text{[Co2]}$$

$$\text{check-operators}(T,\ T_1)\ =\ \text{nil}\quad \textbf{otherwise} \qquad \text{[Co]}$$

4.6.2 Different variables left-hand side equation

The left-hand side of an incremental equation must be left-linear, meaning that all variables should have different names. Let a function on the left-hand side of an equation have two variables: *Var* representing the first child of the constructor and *Var*, representing the second child of the constructor. If a function on the right-hand uses the variable *Var*, it is not possible to determine whether it is the first or the second child of the constructor on the left-hand side of the equation whereas we need this information during incremental reduction. In the module Different-vars given below, we have specified the check that all variables have different names on the left-hand side of an incremental equation.

module Different-vars
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    different-vars(ATerm) → ATerm
hiddens
  context-free syntax
    check-variables(ATerm) → ATerm
equations

The function different-vars below expects as input an ATerm representing an incremental equation. The variables in the left-hand side of the equation, represented by T_1, are selected by means of the function vars that returnes all variables present in the input. To these variables the function check-variables is applied. The function different-vars returns a list of literals representing all variables that occurred more than once on the left-hand side of the equation. If no variables occur more than once, nil will be returned as result of the function.

$$\begin{array}{l}\text{different-vars}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \\ \text{different-vars-msg}(\text{check-variables}(\text{vars}(T_1)))\end{array} \quad \text{[Dv]}$$

The function check-variables below recursively checks if the head of the input list is present in the tail. If the head, a variable, is present in the tail it is returned, together with the result of checking the rest of the list. If no variable occurs more than once nil is returned.

$$\frac{\text{true} = \text{occurs}(T_1,\ T_2)}{\text{check-variables}(T_1;\ T_2) = \text{merge}(T_1, \text{check-variables}(T_2))} \quad \text{[Cv1]}$$

$$\frac{\text{false} = \text{occurs}(T_1,\ T_2)}{\text{check-variables}(T_1;\ T_2) = \text{check-variables}(T_2)} \quad \text{[Cv2]}$$

$$\text{check-variables}(T) \ = \ \text{nil} \quad \textbf{otherwise} \quad \text{[Cv]}$$

4.6.3 Checking the constructors

The first argument of an incremental term occurring in the right-hand side or in the conditions of an equation, has to be a child of the constructor of the left-hand side of the equation. It is also possible that the first argument is equal to the constructor. This occurs when an equation defines an injection. In the module Constructors given below, we have specified a test on these first arguments of incremental terms occurring in an equation.

module Constructors
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    constructors(ATerm) → ATerm

hiddens
context-free syntax
check-constructors(ATerm) $\rightarrow$ ATerm
equations
The function constructors defined below takes an ATerm representing an incremental equation as argument. A check is performed to make sure that the first argument of all terms in the equation (except the top operator of the left-hand side of the equation) is a child of the constructor on the left-hand side of the equation. All incremental terms are selected in the conditions and the right-hand side of the equation by means of function inc-terms. With these incremental terms the function check-constructors is applied.

$$\frac{T = \text{inc-terms(parameters(}T_1\text{); } T_2\text{; } T_3\text{)}}{\begin{array}{l}\text{constructors([scCondEquation } L_1\text{; } L_2\text{; } T_1\text{; } T_2\text{; } T_3\text{]) =}\\ \text{constructors-msg(check-constructors(}T\text{))}\end{array}} \quad [\text{Co}]$$

The function *check-constructors* below checks that the constructors of the functions have been annotated. Prior to this check the function restruct-eqs should have been applied wherein all children of a constructor are annotated with a rank. If this rank is not present we know that the constructor of the term is not a child of the constructor of the left-hand side of the equation. If no annotation is found, a string is returned representing the term, otherwise the function yields nil.

$$\text{check-constructors(}T_1\text{; } T_2\text{)} = \text{merge(check-constructors(}T_1\text{), check-constructors(}T_2\text{))} \quad [\text{Cco1}]$$

$$\begin{array}{lll} \text{check-constructors([[scCfFunction } T\text{] } T_1 \text{ / } T_2\text{; } T_3\text{])} & = \text{nil} & [\text{Cco2}]\\ \text{check-constructors([[scCfFunction } T\text{] } T_1 \text{ / } T_2\text{])} & = \text{nil} & [\text{Cco3}]\\ \text{check-constructors(}T\text{)} & = \text{mk-terms(}T\text{)} \quad \textbf{otherwise} & [\text{Cco}] \end{array}$$

4.6.4 No equations for constructors

According to Definition 2.1 (i), no equations may be defined for the constructors of incremental specifications. In the module No-constructor-eqs shown below we have specified a check to make sure that there are no such equations present in the specification.

module No-constructor-eqs
imports ShortMessages$^{E.2.1}$
exports
context-free syntax
no-constructor-eqs(ATerm, ATerm) $\rightarrow$ ATerm
hiddens
context-free syntax
check-modules(ATerm, ATerm) $\rightarrow$ ATerm
check-equations(ATerm, ATerm) $\rightarrow$ ATerm
equations
We do not allow equations defined for the constructors. The function *no-constructor-eqs* below checks if there are any equations defined for the constructors in a specification. In equation $[NCe]$ given below T_4 is the constructor of a function and T_5 represents the constructors variables. T is a list of ATerms, $(T_i/L_i; \ldots T_n/L_n)/L$ where T_j is the top operator in an equations and L_j is the tag of that equation. L is the name of the module in which these operators and tags occur. We will check that T_4, the constructor of this equation does not occur in the lists of operators by means of the function check-modules.

$$\frac{[T_4 \;\; T_5] = \text{constructor(}T_1\text{)}}{\begin{array}{l}\text{no-constructor-eqs([scCondEquation } L_1\text{; } L_2\text{; } T_1\text{; } T_2\text{; } T_3\text{], } T\text{) =}\\ \text{no-constructor-eq-msg(check-modules(}T_4\text{, } T\text{) / mk-term(}T_1\text{))}\end{array}} \quad [\text{NCe}]$$

The function check-modules enables us to inform the user in which module an eventual equation is defined for the constructor (*nce* is *no constructor equations* abbreviated).

$$\text{check-modules}(T,\ T_1;\ T_2) \ = \ \text{merge}(\text{check-modules}(T,\ T_1), \text{check-modules}(T,\ T_2)) \qquad [\text{Cmod1}]$$
$$\text{check-modules}(T,\ T_1\ /\ L) \ = \ \text{nce-mod-msg}(\text{check-equations}(T,\ T_1)\ /\ L) \qquad [\text{Cmod2}]$$

The function *check-equations* below returns the tags of all equation of which the top operator equal is to the constructor.

$$\text{check-equations}(T,\ T_1;\ T_2) = \qquad [\text{Ceq1}]$$
$$\text{merge}(\text{check-equations}(T,\ T_1), \text{check-equations}(T,\ T_2))$$

$$\text{check-equations}(T,\ T\ /\ L) \ = \ \text{nce-eq-msg}(L) \qquad [\text{Ceq2}]$$
$$\text{check-equations}(T,\ T_1) \ = \ \text{nil} \qquad \textbf{otherwise} \qquad [\text{Ceq}]$$

4.7 Checking the well-presentedness

A specification is well-presented if all the attributes, both function attributes as well as parameter attributes, are uniquely defined by one origin in the equations in that specification, see Section 2.2. In this section we will introduce the method we use to check if a specification is well-presented. The function attributes are always defined by the right-hand side of its defining equation. A parameter term on the right-hand side or in the conditions of an equation, is defined by the origin of the variable or function that it represents. To be able to check if all parameter terms are uniquely defined we need to know how all parameter terms in the specification are defined. If a parameter term, par(ϕ,j) is defined in one place by par(φ,i) and in another place by par(χ,k), we have to check if par(φ,i) and par(χ,k) are defined by the same origin. The check consists of two parts: first information must be gathered concerning the definition rules of all parameters and secondly each definition rule must be compared with all the other definition rules. In Section 4.7.1 we will describe how a table is built containing the definition rules. In Section 4.7.2 we will continue with the actual Well-presentedness check. Below the module Check-WP is shown containing the "global" Well-presentedness check.

module Check-WP
imports WP$^{E.5.2}$ WP-table$^{E.5.1}$
exports
  context-free syntax
    check-wp(ATerm) $\rightarrow$ ATerm
equations

$$\frac{T' = \text{wp-table}(T),\quad T_1 = \text{wp}(T',\ T',\ T')}{\text{check-wp}(T) = \text{wp-msg}(T_1)} \qquad [\text{Cwp1}]$$

4.7.1 WP-table: definition rules for parameter attributes

To be able to perform a well-presentedness check information must be gathered concerning the definition rules of all attributes in a specification. In this section we will describe how appropriate information is gathered and put in a table that can be used when the well-presentedness is checked.

The table is a list of ATerms. Each parameter term in a specification has its own entry in the table and this entry describes how the related attribute is defined. An entry in the table has the form: $[[T_1\ T_2/L]T_3]/(L_1/L_2/L_3)$. In Example 4.2 we show with what information an entry in the table is filled.

Example 4.2 The equation below has been taken from the Pico-typecheck module we discussed in Section 2.3.2.

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) = \mathit{Bool}\end{array}}{\mathsf{tcstat}(\mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od},\ \mathit{Tenv}) = \mathit{Bool}} \quad [\mathsf{Tc10}]$$

The functions tcstat, tcs and tcexpr have all been declared as incremental. For this equation, two entries in the table will be made: one for the parameter attribute of tcexpr and one for the parameter attribute of tcs. In this example we will only discuss the table entry for the parameter attribute in tcs. The entry in the table for tcs is:

[[`"par(tcs,1)"` while Exp do Series od/2] `"par(tcstat,1)"`] / (Tc10/`tcs(Series, Tenv)`/ *Tenv*)

The parameter attribute for the function tcs in the equation is `"par(tcs,1)"` and is represented by the variable *Tenv* in the equation. When the function `tcs(Series, Tenv)` is defined for the second subterm of the constructor (while Exp do Series od/2, 2 indicates that *Series* is the second subterm of while Exp do Series od), `"par(tcs,1)"` is defined by `"par(tcstat,1)"`. The additional information (Tc10/`tcs(Series, Tenv)`/ *Tenv*) will be used for error messages. □

In the module WP-table we have specified how the table discussed above is constructed. The input expected is a list of incremental equations that are the result of the application of the function restruct-eqs we discussed in Section 3.5.5.

module WP-table

imports Restruct-eqs$^{D.6}$
exports
  context-free syntax
    wp-table(ATerm) → ATerm
hiddens
  context-free syntax
    table-per-equation(ATerm) → ATerm
    table-per-term(ATerm, ATerm, ATerm) → ATerm
    table-per-parameter(ATerm, ATerm, ATerm, ATerm, ATerm, INT) → ATerm
equations

The function wp-table constructs a table we will use when we check whether all parameter attributes are uniquely defined. This table contains information about all parameter attributes in the specification and their definition.

$$\mathsf{wp\text{-}table}(T_1;\ T_2) = \mathsf{merge}(\mathsf{wp\text{-}table}(T_1),\ \mathsf{wp\text{-}table}(T_2)) \quad [\mathsf{WPt1}]$$

$$\mathsf{wp\text{-}table}(T\ /\ T_1) = \mathsf{wp\text{-}table}(T) \quad [\mathsf{WPt2}]$$

$$\mathsf{wp\text{-}table}(T) = \mathsf{table\text{-}per\text{-}equation}(T) \quad \textbf{otherwise} \quad [\mathsf{WPt}]$$

table-per-equation: For every equation in the input the following actions are taken. The constructor without variables is selected on the left-hand side of the equation by means of the function constructor. This constructor is represented by the variable T_4 in equation $[Tpe2]$. The function inc-terms selects all incremental terms in the conditions and on the right-hand side of the equation, T_6. With T_4, T_6 and the tag of the equation, L_2 as arguments, the function table-per-term is invoked.

$$\mathsf{table\text{-}per\text{-}equation}(T_1;\ T_2) = \mathsf{merge}(\mathsf{table\text{-}per\text{-}equation}(T_1),\ \mathsf{table\text{-}per\text{-}equation}(T_2)) \quad [\mathsf{Tpe1}]$$

$$\frac{\begin{array}{c}[T_4\ T_5] = \mathsf{constructor}(T_1),\\ T_6 = \mathsf{inc\text{-}terms}(T_2;\ T_3)\end{array}}{\begin{array}{l}\mathsf{table\text{-}per\text{-}equation}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) =\\ \mathsf{table\text{-}per\text{-}term}(L_2,\ T_4,\ T_6)\end{array}} \quad [\mathsf{Tpe2}]$$

table-per-term: For every incremental term the following is done. If the constructor in the term is annotated (T_3/L_3), L_3 is the rank of the constructor of the term, [$Tpt2$], a declaration representation (`"tcs(SERIES,TEN V)->BOOL"`) is made of the term by means of function mk-decl (T_5) and a term representation
(`"tcs(Series,Tenv)"`) is made of the same term by means of the function mk-term (T_6). The function table-per-parameter will now be invoked with the following arguments. The tag of the equation, L, the constructor of the left-hand side of the equation annotated with the rank of the constructor of the term T_1/L_3, the declaration representation of the term T_5, the term representation of the term T_6, the parameters of the term T_4 and a counter "1"; for each parameter in the term an entry in the table must be made. This counter will be used to numerate the corresponding parameter attributes.

$$\text{table-per-term}(L,\ T_1,\ T_2;\ T_3) = \qquad [\text{Tpt1}]$$
$$\text{merge}(\text{table-per-term}(L,\ T_1,\ T_2),\ \text{table-per-term}(L,\ T_1,\ T_3))$$

$$\frac{\begin{array}{l} T_5 = \text{mk-decl}([\text{scCfFunction}\ T_2]), \\ T_6 = \text{mk-term}([[\text{scCfFunction}\ T_2]\ T_3\ /\ L_3;\ T_4]) \end{array}}{\begin{array}{l} \text{table-per-term}(L,\ T_1,\ [[\text{scCfFunction}\ T_2]\ T_3\ /\ L_3;\ T_4]) = \\ \text{table-per-parameter}(L,\ T_1\ /\ L_3,\ T_5,\ T_6,\ T_4,\ 1) \end{array}} \qquad [\text{Tpt2}]$$

$$\text{table-per-term}(L,\ T_1,\ T_2)\ =\ \text{nil}\quad \textbf{otherwise} \qquad [\text{Tpt}]$$

Equation [$Tpp1$] will be applied as long as there is more than one parameter in the list of parameters, and the counter will be increased by one.

$$\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4;\ T_4',\ \mathit{Int}) = \qquad [\text{Tpp1}]$$
$$\text{merge}(\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4,\ \mathit{Int}),$$
$$\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4',\ \mathit{Int}+1))$$

Equation [$Tpp2$] is applied if the parameter is annotated T_4/L_4. If the parameter is defined by a parameter on the left-hand side of the equation this parameter has been annotated during the restruction of the equation. T_4 is the parameter attribute representation of the parameter on the left-hand side of the equation. In this case we make a parameter attribute representation of the term declaration T_2 combined with the counter Int by means of the function cc (concatenate) and the result will be, for instance, `"par(tcs(SERIES,TENV)->BOOL,1)"`; T_5. Of the variable that represents the parameter, a term representation is made by means of function mk-term, for instance `"Tenv"`;T_6. The variable T_1 is the constructor that was annotated in equation [$Tpt2$], T_3 is the term represention of the term. These objects are now put in a new entry of the table.

$$\frac{\begin{array}{l} T_5 = \text{par-att}(T_2,\ \mathit{Int}), \\ T_6 = \text{mk-term}(T_4) \end{array}}{\begin{array}{l} \text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4\ /\ L_4,\ \mathit{Int}) = \\ [[T_5\ T_1]\ L_4]\ /\ (L\ /\ T_3\ /\ T_6) \end{array}} \qquad [\text{Tpp2}]$$

Equation [$Tpp3$] applies if the parameter term is not annotated ($[T_4\ T_4']$). In this case it is not defined by a parameter term on the left-hand side of the equation but probably by a constant or a term. In this case, a string representation is made of the parameter term. T_6 represents both the definition of the parameter attribute as well as the string corresponding to the parameter attribute.

$$\frac{\begin{array}{l} T_5 = \text{par-att}(T_2,\ \mathit{Int}), \\ T_6 = \text{mk-term}([T_4\ T_4']) \end{array}}{\begin{array}{l} \text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ [T_4\ T_4'],\ \mathit{Int}) = \\ [[T_5\ T_1]\ T_6]\ /\ (L\ /\ T_3\ /\ T_6) \end{array}} \qquad [\text{Tpp3}]$$

$$\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4,\ \mathit{Int})\ =\ \text{nil}\quad \textbf{otherwise} \qquad [\text{Tpp}]$$

4.7.2 WP: comparing definitions

In the module WP given below the actual well-presentedness check is performed.

module WP
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    wp(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    wp5(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
    wp6(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
    compare-definitions(ATerm, ATerm) $\rightarrow$ ATerm
    last(ATerm) $\rightarrow$ ATerm
  variables
    Entry [0-9]$*$ $\rightarrow$ ATerm
    Entries $\rightarrow$ ATerm
    Par [0-9]$*$ $\rightarrow$ ATerm
    Def [0-9]$*$ $\rightarrow$ ATerm
    Tab [0-9]$*$ $\rightarrow$ ATerm
    Info [0-9]$*$ $\rightarrow$ ATerm
    Def [0-9]$*$ $\rightarrow$ ATerm
    TransDef [0-9′]$*$ $\rightarrow$ ATerm
    ParConstr $\rightarrow$ ATerm
    Constr $\rightarrow$ ATerm
equations

Originally, all three arguments of the function wp are the same. They are all a copy of the table produced in the module WP-table and contain the definition rules for all parameter attributes in a specification. Every entry in the first table is compared with every entry in the second table by means of function wp5. The third copy of the table remains intact. The function wp5 is applied to check if property (v) in Definition 2.1 holds: two identical parameters defined for the same constructor must be defined by the same parameter term, constant or function.

$$\mathsf{wp}(\mathit{Entry};\ \mathit{Entries},\ \mathit{Tab}_1,\ \mathit{Tab}_2) = \mathsf{merge}(\mathsf{wp}(\mathit{Entry},\ \mathit{Tab}_1,\ \mathit{Tab}_2),\ \mathsf{wp}(\mathit{Entries},\ \mathit{Tab}_1,\ \mathit{Tab}_2)) \quad \text{[WP1]}$$

$$\mathsf{wp}(\mathit{Def}\ /\ \mathit{Info},\ \mathit{Tab}_1,\ \mathit{Tab}_2) = \mathsf{wp5}(\mathit{Def}\ /\ \mathit{Info},\ \mathit{Tab}_1,\ \mathit{Tab}_2) \quad \text{[WP2]}$$

The function wp5 defined below checks if two parameters with the same name and defined for the same child of a constructor, are defined equally. If the two parameters are not defined by the same term (equation [$WP53$]), the transitive definition will be found for both of these parameters by means of function wp6. These two transitive definition will be compared by means of function the function compare-definitions. The equation [$wp52$] is applied if two attributes are defined by the same term but occur in two different equations. We do not have to check parameters that are defined equally.

$$\mathsf{wp5}(\mathit{Entry},\ \mathit{Entry}_1;\ \mathit{Entries},\ \mathit{Tab}_2) = \mathsf{merge}(\mathsf{wp5}(\mathit{Entry},\ \mathit{Entry}_1,\ \mathit{Tab}_2),\ \mathsf{wp5}(\mathit{Entry},\ \mathit{Entries},\ \mathit{Tab}_2)) \quad \text{[WP51]}$$

$$\mathsf{wp5}(\mathit{Def}\ /\ \mathit{Info}_1,\ \mathit{Def}\ /\ \mathit{Info}_2,\ \mathit{Tab}_2) = \mathsf{nil} \quad \text{[WP52]}$$

$$\frac{\begin{array}{r}Par_1 \neq Par_2,\\ TransDef_1 = \mathsf{wp6}(Par_1,\ Tab,\ Tab),\\ TransDef_2 = \mathsf{wp6}(Par_2,\ Tab,\ Tab),\\ TransDef_1 \neq \mathsf{nil},\\ TransDef_2 \neq \mathsf{nil},\\ TransDef_1{}' = \mathsf{merge}(Info_1,\ TransDef_1),\\ TransDef_2{}' = \mathsf{merge}(Info_2,\ TransDef_2)\end{array}}{\begin{array}{l}\mathsf{wp5}([ParConstr\ Par_1]\ /\ Info_1,\ [ParConstr\ Par_2]\ /\ Info_2,\ Tab) =\\ \mathsf{compare\text{-}definitions}(TransDef_1{}',\ TransDef_2{}')\end{array}} \quad \text{[WP53]}$$

$$\mathsf{wp5}(Entry_1,\ Entry_2,\ Tab) \ = \ \mathsf{nil} \quad \textbf{otherwise} \qquad \text{[WP5]}$$

The function wp6 defined below searches for the transitive definition of a parameter attribute. When an attribute is found, the function wp6 is applied to the definition of that attribute. This goes on until an attribute is defined by a term or a constant, and the search is completed. This term or constant originally defines the attribute we started with and all the other attributes in the transitive definition for that matter. A transitive definition looks like: `"par(tcstat,1)"`;`"defined by"`; `"par(tcs,1)"`;`"defined by"`;`"tcd(Decls)"`.

$$\frac{Par \neq Par_1}{\begin{array}{l}\mathsf{wp6}(Par,\ [[Par\ Constr]\ Par_1]\ /\ Info;\ Entries,\ Tab) =\\ \mathsf{merge}(Info,\ \mathsf{wp6}(Par_1,\ Tab,\ Tab))\end{array}} \quad \text{[WP61]}$$

$$\frac{Par \neq Par_1}{\begin{array}{l}\mathsf{wp6}(Par,\ [[Par_1\ Constr]\ Par_2]\ /\ Info;\ Entries,\ Tab) =\\ \mathsf{wp6}(Par,\ Entries,\ Tab)\end{array}} \quad \text{[WP62]}$$

$$\begin{array}{l}\mathsf{wp6}(Par,\ [[Par\ Constr]\ Par]\ /\ Info;\ Entries,\ Tab) =\\ \mathsf{wp6}(Par,\ Entries,\ Tab)\end{array} \quad \text{[WP63]}$$

$$\frac{Par \neq Par_1}{\mathsf{wp6}(Par,\ [[Par\ Constr]\ Par_1]\ /\ Info,\ Tab) = Info} \quad \text{[WP64]}$$

$$\frac{\begin{array}{c}Par \neq Par_1,\\ Par \neq Par_2\end{array}}{\mathsf{wp6}(Par,\ [[Par_1\ Constr]\ Par_2]\ /\ Info,\ Tab) = \mathsf{nil}} \quad \text{[WP65]}$$

$$\mathsf{wp6}(Par,\ [[Par\ Constr]\ Par]\ /\ Info,\ Tab) \ = \ Info \qquad \text{[WP66]}$$

$$\mathsf{wp6}(Par,\ Tab_1,\ Tab_2) \ = \ Par \quad \textbf{otherwise} \qquad \text{[WP6]}$$

The auxiliary function compare-definitions defined below removes all the transitive definitions that are equal or has the same constant or term as last object. If two transitive definitions are not removed they are returned and an error message can be created, containing the whole definition path to make the user understand what went wrong.

$$\mathsf{compare\text{-}definitions}(T,\ T) \ = \ \mathsf{nil} \qquad \text{[CompD1]}$$

$$\mathsf{compare\text{-}definitions}(T,\ T') \ = \ \mathsf{nil} \quad \textbf{when} \quad \mathsf{last}(T) = \mathsf{last}(T') \qquad \text{[CompD2]}$$

$$\mathsf{compare\text{-}definitions}(T,\ T') \ = \ [T\ T'] \quad \textbf{otherwise} \qquad \text{[CompD]}$$

The function last defined below simply returns the last object in transitive definition.

$$\mathsf{last}(T;\ T_1) \ = \ \mathsf{last}(T_1) \qquad \text{[Last1]}$$

$$\mathsf{last}(T) \ = \ T \quad \textbf{otherwise} \qquad \text{[Last]}$$

4.8 Error messages

Simultaneously with the PRS-check we described in the previous sections information is gathered for error messages. We have structured these message in such a manner that all errors that where found in non-incremental equations are given first followed by messages concerning the incremental equations. The messages regarding the well-presentedness are the last ones on the list. All these three different kinds of messages are given per module and per equation with a brief description of the reason for the error. The messages have been defined in the module ShortMessages that can be found in Appendix E.2.1.

Example 4.3 This messages was given when the strange type checker given in Example 2.3 was PRS-checked. The imported module Pico-typecheck was not part of the input:

```
"The module(s) cannot be transformed"    ;
"Module : Strange-typechecker"    ;
" Eq:[Stc1] :"    ;
"Incremental functions not declared in checked specification :"  ;
"tcd(Decls)"    ;
" Eq:[Stc2] :"    ;
"Incremental functions not declared in checked specification :"  ;
"tcstat(Stat,Tenv1)"    ;
" Eq:[Stc3] :"    ;
"Incremental functions not declared in checked specification :"  ;
"tcstat(Stat,Tenv)"    ;
"WP-error, some parameter-terms are not uniquely defined"    ;
"Eq:[Stc3] tcstat(Stat,Tenv) Parameter: Tenv"    ;
 "defined by"    ;
  "Eq:[Stc1] stcs2(Series,tcd(Decls)) Parameter: tcd(Decls)"  ;
    "defined by"  ;
      "tcd(Decls)" ;
"Eq:[Stc2] tcstat(Stat,Tenv1) Parameter: Tenv1"   ;
 "defined by"    ;
   "Eq:[Stc1] stcs1(Series,(x:natural)+empty-tenv)
                             Parameter: (x:natural)+empty-tenv"
```

□

We have also specified longer versions of the error message in the module LongMessages. This module can be found in Appendix E.2.2. We have defined three functions in this module. If the function add-long-message is invoked with as argument a message with short messages, long messages corresponding to the short messages occurring in the message will be added at the beginning of the list of messages. When the function get-long-msg is invoked with the same input, only the long messages are returned. There is also a function long-msg. If this function is applied with a short message its corresponding long message is returned. This last function can be used in an interactive environment.

4.9 Conclusions

We have specified in ASF+SDF a check on a list of ASF+SDF modules in ASFIX notation for checking if the modules form a Well-presented PRS. Simultaneously with the check, information is gathered to enable detailed error messages. The messages that are generated are brief but given these messages, longer and more detailed messages can be generated. The method for incremental rewriting presented in this thesis does not cover the case when the constructor is defined by a list operator. In the rest of this thesis we will assume that all specifications are well-presented primitive recursive schemes with parameters with no list operators in the constructors.

Chapter 5

Storage for incremental reduction

5.1 Introduction

During incremental reduction with incremental specifications normal forms are stored in attributes in a storage that corresponds to the abstract syntax tree in an editor. When the term is re-evaluated, the normal forms in the storage that are not dependent on changes that might have been made in the term can be retrieved and reused. In this chapter we will describe the storage we have defined for this purpose. The storage we have defined consists of nodes, attributes and storage directions. In Section 5.2 we will describe these different components. Section 5.3 is devoted to a special case concerning *non incremental children*. In Section 5.4 it will become clear why we have introduced the storage directions and in Section 5.5 we will discuss how injections influence the shape of the storage.

5.2 The storage structure

We have defined a storage structure that simulates a tree. We will refer to this tree as the *storage-tree*. The storage structure consists of three parts: the storage directions, a list of nodes and a list of attributes. We will start with a description of the nodes, followed by the definition of the storage as a whole. We close this section by introducing the attributes we use to store and retrieve normal forms during incremental reduction.

5.2.1 Nodes

The nodes in a storage structure represents the abstract syntax tree T of a text in an editor. Each node N_s in the storage corresponds to a node N_{ast} in this abstract syntax tree T_{ast}. To be able to distinguish the nodes in the storage, we give them a unique *node number*. Nodes in the storage have references to their children. Assume that N_i is a node in the storage and that N_i corresponds to a node N_{ast} in the abstract syntax tree. If N_{ast} has two children, N'_{ast} and N''_{ast}, and the two nodes N_j and N_k in the storage correspond to N'_{ast} and N''_{ast} respectively, then N_j and N_k are children of N_i. The children of a node are represented by a list of node numbers in the node. Thus the list of children in N_i above is: *j;k*. The order of this list is important. We say that N_j is the first child of N_i and N_k is its second child. A node which node number occur on the *r*'s position in a list of children has *rank r*.

Example 5.1 In Figure 5.1 the relation between an abstract syntax tree and the storage-tree is shown. We have connected the nodes in the storage with their children. The dotted arrows between the nodes in the abstract syntax tree and the storage-tree indicate the correspondence between the nodes. N_1 has two children, N_2 and N_3 and its list of children is thus: 2;3. N_3 also has two children, N_4 and N_5 and is lists of children is: 4;5. N_2, N_4 and N_5 do not have any children in this example and their list of children is empty. □

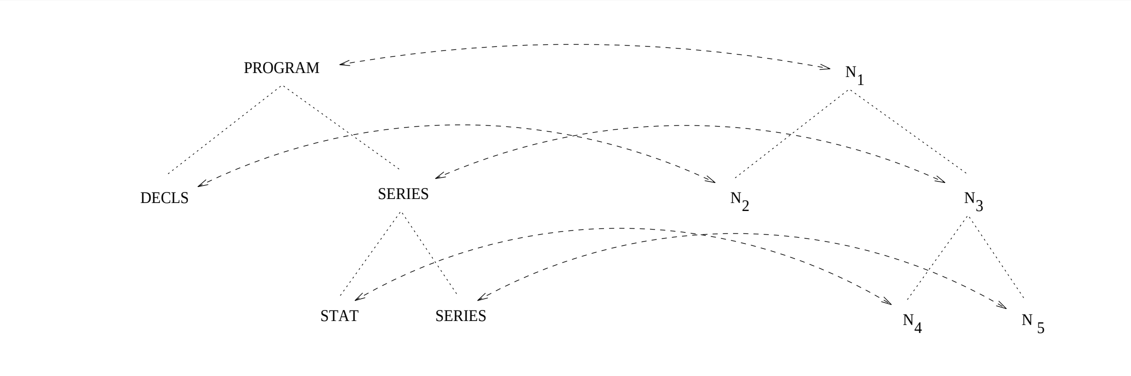


Figure 5.1: An abstract syntax tree and its corresponding storage-tree.

We have specified the nodes in the module Nodes shown below.

module Nodes
imports Integers$^{A.3}$
exports
  sorts NODE-PTR NODE-PTRS STATE NODE-CONTENT NODE NODES
  context-free syntax

INT	→ NODE-PTR
{INT ";"}*	→ NODE-PTRS
init	→ STATE
update	→ STATE
non-inc-child-changed	→ STATE
top-new-subtree	→ STATE
idle	→ STATE
cur-state STATE children {INT ";"}*	→ NODE-CONTENT
NODE-PTR ":" NODE-CONTENT	→ NODE
{NODE ";"}*	→ NODES

A node consists of two parts: a unique node number and the node content.

The *node number* simulates a pointer; we suggest that the unique node references should be replaced by pointers when the storage structure later is implemented in e.g. C. The node content consists of a cur-state, the current state and a list of the node's children.

The five STATE's a node can have (init, update, non-inc-child-changed, top-new-subtree and idle) are only used when the the storage is initialised or updated after changes have been made to the abstract syntax tree T_{ast} to which the storage T_s corresponds. We postpone the explanation of these different states until we discuss the functions defined on the storage during initialisation in Section 7.3. Functions defined on the nodes have been defined in the module Node-functions that can be found in Appendix F.2.

5.2.2 The storage

During incremental reduction, it is important that a node in the storage can be found without time consuming search actions. This can be accomplished if a reference is maintained to the *current node* in the storage. With current node we mean a node N_s in the storage that corresponds to a node N_{ast} in the abstract syntax such that N_{ast} is the node that, at that point, is being evaluated. Some additional information must be maintained in order to keep track of the current node. In the module Storage shown below we have defined a structure to maintain this information.

module Storage
imports Attribute-functions$^{F.4}$
exports
  sorts STORAGE S-DIRECTIONS
  context-free syntax
    s-top NODE-PTR
    cur NODE-PTR
    max INT
    stack {INT ";"}*
    level INT
    path {INT ";"}*
    ch-atts {ATT-PTR ";"}*   → S-DIRECTIONS
    S-DIRECTIONS ":" {NODE ";"}*
    ":" {ATTRIBUTE ";"}*   → STORAGE

The storage is split in three parts: the storage directions, a list of nodes and a list of attributes. We will now describe the storage directions.

The s-top is the number of the node that is the top of the storage tree. If the storage is empty, the top will be given the value 0, our nil pointer. The cur is the number of the current node in the storage. To be able to keep track of the current node N_s in the storage during reduction and initialisation of the storage, we have defined the stack and the level. How we keep track of this current node by means of the stack and the level will be described in Section 5.4.

When an abstract syntax tree T_{ast} is changed, parts of the storage might need to be updated or initialised. The path in the storage content should be set before the updating of the storage begins and should be supplied by an external function. The path describes the route that should be followed from the top of the storage to the node that must be updated. The path is a list of ranks. If the first rank on the path is j, the new current node should be the j's child of the current node. This first rank on the path should then be remove from the path and the process should be continued with the adjusted path.

ch-atts is a list of references to attributes that have been changed during incremental reduction of a term. In Section 7.2.3 the changed attributes will be discussed in detail. Functions defined on the storage structure have been specified in the module Storage-functions that can be found in the Appendix F.6

5.2.3 Attributes

Attributes can be thought of as labels attached to nodes in the storage and are used for storing results when terms are applied to functions in an incremental specification. We have defined the attributes in the module Attributes shown below.

module Attributes
imports Node-functions$^{F.2}$
exports
  sorts ATT-NAME ATT-NAMES NORMAL-FORM ATT-TYPE STATUS
    ATT-PTR ATT-PTRS ATT-CONTENT ATTRIBUTE
    ATTRIBUTES
  lexical syntax
    "\"" ~[\n\"]* "\"" → ATT-NAME
  context-free syntax
    {ATT-NAME ";"}+   → ATT-NAMES

```
"|" empty "|"               → NORMAL-FORM

fun                         → ATT-TYPE
par                         → ATT-TYPE

"IN"                        → STATUS
"TE"                        → STATUS
"C"                         → STATUS
"UC"                        → STATUS
safe                        → STATUS
unsafe                      → STATUS

INT ":" {ATT-NAME ";"}+     → ATT-PTR
{ATT-PTR ";"}*              → ATT-PTRS

type ATT-TYPE
status STATUS
nf NORMAL-FORM
preds {ATT-PTR ";"}*
succs {ATT-PTR ";"}*
corr-f-a {ATT-PTR ";"}*     → ATT-CONTENT
ATT-PTR ":" ATT-CONTENT     → ATTRIBUTE
{ATTRIBUTE ";"}*            → ATTRIBUTES
```

The name of an attribute is a quoted string or a list of quoted strings, ATT-NAMES. If the name of a function is ϕ, we use "ϕ" as the name of the function attribute. A parameter attribute is named by "par(ϕ,k)" when it is associated with the k's parameter of ϕ. If some parameter attributes are defined equally, only one attribute will be created given the names of the parameter attributes it represents. The name of an attribute combined with the number of the node to which it is attached is a unique combination and attributes can be distinguished. We refer to this combination as an attribute pointer, ATT-PTR.

Initially, we have defined the sort NORMAL-FORM as the constant empty between bars. When a specification is transformed into an incremental specification, we will add the result sort of each ϕ-function in the specification to the definition of the sort NORMAL-FORM by defining that each of these result sorts between bars are also NORMAL-FORMs.

To be able to efficiently determine whether an attribute is a parameter attribute or a function attribute, we introduce the sort ATT-TYPE with two constants: fun for function attributes and par for parameter attributes.

When an attribute that has status "Initial" (IN) or "TobeEvaluated" (TE), the value in the attribute is not correct and we refer to the attribute as *unsafe*. We say that an attribute is *safe* if its value is correct. In this case the status of the attribute is "UnChanged" (UC) or "Changed" (C). For efficiency reasons that will be explained in Section 7.2.3 we use four different status indications rather than just safe and unsafe.

Both function attributes and parameter attributes have a list of references to their predecessor preds and successors succs, but only attributes of type par have a list of references to their corresponding function attributes, corr-f-a. How the predecessors and successors of attributes can be derived is described below.

The reason why we have defined the attributes as two parts, an attribute pointer combined with the content of an attribute is that the equations defining functions such as find-att can be written in ASF+SDF with less complexity.

The functions on attributes and their defining equations have been specified in the module Attribute-functions that can be found in Appendix F.4.

Attribute dependencies

In Section 2.4.3 we discussed how attributes are defined. If an attribute att_1 is *defined by* another attribute att_2, a dependency runs from att_2 to att_1: $att_2 \rightarrow att_1$. In this case, att_2 is a predecessor of att_1 and att_1 is a successor of att_2. An algorithm for deriving attribute dependencies in an equation is given in [Meu94].

Example 5.2 The attributes that corresponds to the equation given below are: tcs, tcstat, par(tcs,*1*) and par(tcstat,*1*).

$$\mathsf{tcs}(\mathit{Stat},\ \mathit{Tenv})\ =\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv}) \qquad \text{[Tc6]}$$

The function attribute tcs depends on the value of tcstat, thus a dependency runs from tcstat to tcs: tcstat $\rightarrow$ tcs. The value of *Tenv* on the right-hand side of the equation depends on the value on *Tenv* on the left-hand side of the equation. Therefore a dependency runs from par(tcs,*1*) to par(tcstat,*1*): par(tcs,*1*) $\rightarrow$ par(tcstat,*1*). tcs is a successor of tcstat and tcstat is a predecessor of tcs. par(tcs,*1*) is a predecessor of par(tcstat,*1*) and par(tcstat,*1*) is a successor of par(tcs,*1*). □

In addition to the dependencies derived from equations, transitive dependencies are used between parameter attributes and their *corresponding* function attributes. The corresponding attribute of par(ϕ,k) is the function attribute ϕ attached to the same node. If the name of an parameter attribute is a list par(φ,i);par(χ,j), then its corresponding attributes are φ and χ. These dependencies reflect that the normal form of a Φ-term depends on the value of its parameter terms. These dependencies will be used as short cuts when the status of the attributes are updated.

5.3 Non incremental children

In Section 5.2.1 we defined the nodes we use in the storage-tree and we also showed the mapping between the storage-tree and the abstract syntax tree. We will now describe the nodes in the abstract syntax tree for which there will be no corresponding nodes in the storage-tree. If no ϕ-functions are declared for the sort of a node in the abstract syntax tree that node will not have a corresponding node in the storage-tree. We call such a node a "non-incremental node". According to the definition of a PRS its underlying tree is built from non-incremental nodes as well: a "non-incremental tree". In the storage we will not create nodes corresponding non-incremental nodes. It is, however, convenient to know that these nodes exists in the abstract syntax tree. Assume that a node N_{ast} in the abstract syntax tree has two children, N'_{ast} and N''_{ast} and no ϕ-functions are defined for N'_{ast}. In the storage-tree a node N_i will correspond to N_{ast} and a node N_j to N''_{ast}. The list of children in N_i will then be: 0;j. Thus if a child is referenced by "0" in the list of children, no ϕ-functions have been declared for this subtree in the abstract syntax tree. This is essential when the storage is updated. If a "non incremental" child has been changed, we only need to adjust the status of the attributes.

Example 5.3 In Figure 5.2 the storage-tree and its corresponding abstract syntax tree is shown. In this example, the storage has been initialised for the type checker for a Pico-program. We discussed this type checker in Section 2.3.2. Note that nodes in the abstract syntax tree for which no incremental functions have been declared, do not have any corresponding nodes in the storage. There are no ϕ-functions (incremental functions) defined for the sorts: ID, TYPE and ID-TYPE. □

5.4 Keeping track of the current node

When a term is applied to a ϕ-function in an incremental specification it is important that the current node N_s in the storage corresponds to a node N_{ast} in the abstract syntax tree for the term T, that is at that point being evaluated. By maintaining a reference to the current node, time consuming search operations can be avoided. To be able to maintain the mapping between a node N_{ast} in the abstract syntax tree and the current node N_s in the storage, information is

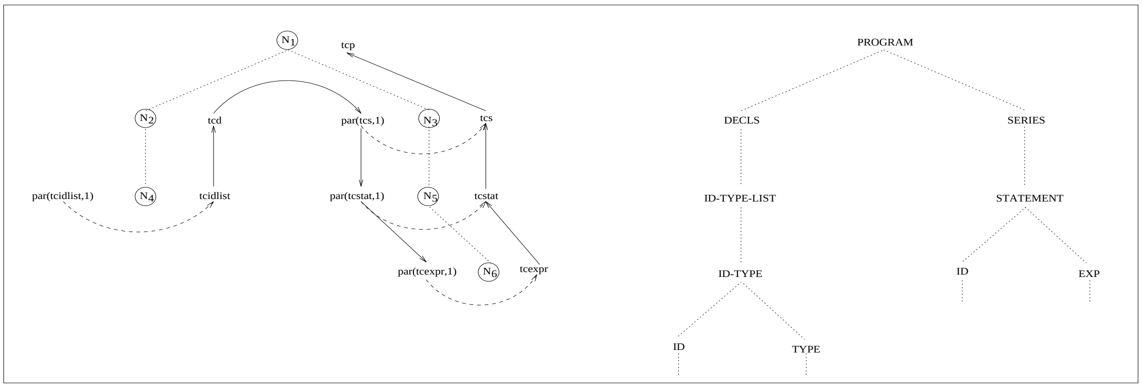


Figure 5.2: The storage-tree with attributes and dependencies added and its corresponding abstract syntax tree.

needed from the rewrite engine during the reduction. When a specification is transformed into an incremental specification, all equations defining incremental functions will be bound by a new first condition and a new right-hand side. The function in the first condition, status-is-safe or status-is-unsafe, will supply the storage with information, *IncInfo*, to set the new current node N'_s in the storage corresponding to the node N_{ast} for which a match has just been found with the left-hand side of an equation. The *IncInfo* consists of two parts: a node rank *R* and a level *L*.

Example 5.4 Before incremental rewriting with incremental specifications can start, a specification must be transformed into one. The equation [Tc7] represents an equation before it has been transformed. Assume that the functions tcs and tcstat have been declared as incremental.

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv})\ =\ \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad \text{[Tc7]}$$

When a specification is transformed, three new equations will be derived from each equation. The results of transforming equation [Tc7] is shown below.

$$\mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ S)\ =\ \mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ 1:0,\ S) \qquad \text{[AII4]}$$

$$\frac{\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}safe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R:L,\ S)}{\begin{array}{l}\mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ R:L,\ S) = \\ \mathsf{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ S_1)\end{array}}\quad \textbf{otherwise} \qquad \text{[SIS4]}$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R:L,\ S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\ \mid \mathit{Tenv} \mid,\ S_1),\\ \mid \mathit{BOOLvar}_1 \mid : S_3 = \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv},\ 1:L+1,\ S_2),\\ \mid \mathit{BOOLvar}_2 \mid : S_4 = \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv},\ 2:L+1,\ S_3)\end{array}}{\begin{array}{l}\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv},\ R:L,\ S) = \\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\\ \qquad \mid \mathit{BOOLvar}_1\ \&\ \mathit{BOOLvar}_2 \mid,\ S_3)\end{array}} \qquad \text{[ITc7]}$$

In equation [AII4] the *IncInfo* is initialised with 1:0. This means that *R* has value 1 when the first equation is applied. The level *L* is set to 0. Equation [SIS4] applies when the value stored in the attribute `"tcs(SERIES,TENV)->BOOL"` is *safe*. When the function tcstat is applied in equation [ITc7] the node rank will be 1; *Stat* is the first child of *Stat;Series*. When tcs is applied, the node rank has value 2; the second child. The level *L* is increased by one when an incremental function is applied (tcstat and tcs). [1] □

[1] One may ask what the variable *S* is in the equations. As a temporary solution to the lack of external functions the storage is part of the term being reduced. See Chapter 6

When a new current node in the storage has been found and set, a reference to this node will be pushed on the stack. Every time something is pushed on the stack, the level L' in the storage is increased by one. When the stack is popped, the level L' will be decreased by one. The level L' in the storage is thus a counter for the stack containing the number of node references on the stack. A reference to the previous current node is present on the stack.

When an equation defining a ϕ-function is matched, the variable L in the equation will be instantiated with the correct level value. If another ϕ-function is invoked in this equation, L will be increased by one and be one of the arguments this function will get. By increasing the variable L by one when a ϕ-function is applied in an equation, we can keep track of the current node in the storage.

Every time a match with an incremental equation is found, the first condition will be tried. In this first condition, the functions status-is-safe or status-is-unsafe ensures that a new current node will be set. Before the new current node is set, the level L in the "specification" and level L' in the storage are compared. If they are equal, the R'th child of the node on top of the stack can be set as new current. When L and L' are not equal, the stack must be popped until they are equal and the new current node can be set.

5.5 Injections

It is not always possible to maintain a mapping between the abstract syntax tree and storage-tree. If injections are present in the specification ad hoc measures must be taken. An injection is in

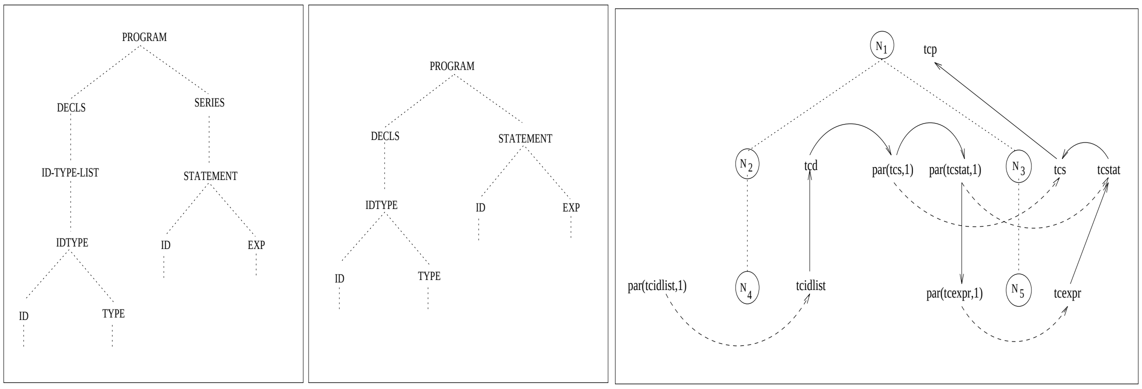


Figure 5.3: The abstract syntax tree with injections and its corresponding tree in the ASF+SDF system and the storage-tree with attributes and dependencies.

this context a function that turns one sort into another sort. In the context-free syntax section of module Pico-syntax in Section 2.3.1 we find the function:

STATEMENT $\rightarrow$ SERIES

This means that a single STATEMENT is also a SERIES of statements. In abstract syntax trees these injections do not occur. The consequence for us is that when Φ-functions are defined for injections, we must "lift" the attributes at a node that do not really exist, to the node's parent. The equation below shows the definition of such a Φ-function.

$$\text{tcs}(\mathit{Stat}, \mathit{Tenv}) = \text{tcstat}(\mathit{Stat}, \mathit{Tenv}) \qquad [\text{Tc6}]$$

Typical for such function is that the first argument of the function on the left-hand side of the equation looks the same as the first argument of a function in the conditions or on the right-hand side of the equation. In Figure 5.3 the relationship between the abstract syntax tree with injections, how this abstract syntax tree is represented in the ASF+SDF system and the effects it has on the storage-tree.

Injections will be detected during the transformation of the modules. To solve the problem with the injections during reduction, "injections" (they do not have a node of their own) are given rank 0. This indicates that the new current node will be set as the previous current node.

5.6 Conclusions

We have specified a storage structure in ASF+SDF that can be used for incremental reduction with incremental specifications. Before reduction starts, the storage must be initialised. An initialised storage reflects the shape of the abstract syntax tree of the term it was initialised for. With a minimum of information supplied to the storage (by means of external functions), a reference is maintained to a current node in the storage. This current node corresponds to the top of the subtree in the abstract syntax tree that is being evaluated at that time. Time consuming search actions are thus avoided. The complexity of the storage tree is linear with respect to the number of nodes in its corresponding abstract syntax tree for which incremental functions have been declared. We are convinced that the specified storage for efficiency reasons should be implemented in a conventional programming language as soon as the use of external functions is supported by the ASF2C-compiler.

Chapter 6

Transforming and deriving modules

6.1 Introduction

Incremental reduction of a term means that computed values are stored and reused. In Chapter 5 we defined a storage for incremental reduction of incremental specifications. The storage needs information during reduction to be able to efficiently determine whether a value stored in an attribute is reliable. In this chapter we will describe how ASF+SDF modules in ASFIX notation can be transformed in such a manner that functions are called in the conditions to manipulate the storage. We assume that the modules that are to be transformed at a previous stage have passed the PRS-check we described in Chapter 4.

To accomplish incremental rewriting with incremental specification three transformations are needed. One to transform a module into an incremental module in which functions are called during reduction to check the status of attributes and to store in and retrieve values from the storage.

A module InitUpdate is needed for initialisation and update of the storage during editing. When a term in a term editor has been edited and a subtree replacement has taken place, the storage corresponding to the "old" term can be updated by applying the function init to the new term. This function is defined in the derived module InitUpdate. When this function is applied functions are called to create new nodes in the storage and label these nodes with new attributes, add dependencies between attributes and remove parts of the storage that are no longer needed. To be able to determine what parts of the storage need updating a module PathSearch should also be derived. The module PathSearch is used to check what part of a term has been changed after editing.

Due to the fact that the external functions we intend to use to manipulate the storage are not yet supported by the rewrite system, we have specified two different versions of transformations. One for testing and one that should be used when the use of these functions is supported and the storage structure is implemented as intended in a conventional programming language such as C. We refer to the version of incremental specification we derive for testing as the ASF+SDF version and to the version that can be used later when the use of external functions is supported as the *external* version.

Side effects are not possible in a rewrite system. For incremental evaluation, side effect are needed and must therefore be simulated. In the ASF+SDF version this is done by letting the storage be part of the term that is being reduced. When an incremental function is applied, the storage will be one of its arguments. The storage is also part of the return value of the function. Of course, this is not efficient but it enables us to test the incremental specifications.

As we mentioned earlier, functions manipulating the storage are invoked in the conditions of a transformed equation. One of the functions added to the conditions is the function status-is-safe.

When the external functions are supported, this function can probably be defined as follows:

$$\frac{\mathit{Bool} = \text{EXTC status\_is\_safe(AN,IncInfo,get\_storage())}}{\text{status-is-safe(AN, }\mathit{IncInfo}\text{)} = \mathit{Bool}} \quad \text{[SIS]}$$

In this case the function status-is-safe does not need the storage as an argument; the function get-storage will locate it and because side effects are possible, the storage does not need to be returned. It is sufficient to return a Boolean value. The code after "EXTC" is C-code.

The transformation of a module into an incremental module for the ASF+SDF version is more complex than for the external version. Every time an incremental function is applied, the storage must be passed as argument and returned as part of the return value. The effect for the transformation is that variables with new names must be used for every new copy of the storage.

Because the transformation of a module into an incremental module assuming that the use of external functions is supported, is less complex we start with a description of this transformation in Section 6.2. In Section 6.3 the differences between the generation of an incremental module for the external version of an incremental module and the ASF+SDF version will be given. How an InitUpdate module can be derived from modules is described in Section 6.4. This description is for the external version. In Section 6.5 we will discuss how an external version of the InitUpdate module can be transformed into an InitUpdate module for the ASF+SDF version. The derivation of a PathSearch module is described in Section 6.6. The module PathSearch is independent of the storage thus the derived module can be used in both versions of incremental specifications.

We have specified the transformations and derivations in ASF+SDF. The modules that are transformed should be in ASFIX notation. As we mentioned in Chapter 3, ASFIX is not meant for the human eye. The equations defining some parts of the transformation are rather ugly. The results of the transformations and derivations are ASFIX-terms representing ASF+SDF modules. These ASFIX modules can be converted into ASF+SDF modules at unix-level by the existing tool ASFIX2ASDF. In the examples throughout this chapter we use parts of the Pico-typecheck specification that was discussed in Section 2.3.2.

6.2 Generation of an incremental module

In this section we will give an informal definition of the transformation performed on a module to create an incremental module. We suppose that the use of external functions is supported and that the storage does not have to be an argument in the incremental functions and part of the return values yielded by these functions. We have tried to specify this transformation so that only the necessary changes are made and the difference between the transformed module and the original module is small. This transformation is defined in the module Mod2IncMod that can be found in Appendix G.2.1. The result of transforming the module Pico-typecheck we described in Section 2.3.2 can be found in the Appendix I.1.

6.2.1 Changing the module name

The original module is not really transformed; a new incremental module is generated. We prefer, however, to refer to this generation as a transformation. To avoid that the original module is replaced by the generated, incremental module, the name of the module is adjusted. The name of the "transformed" module is constructed from the original module name by adding the prefix `Inc` to the module name. It is possible that a module that is being transformed imports other transformed module. Because of the name change, the names of imported modules must also be adjusted. When the list of imported modules is transformed, a list of the names of other, transformed modules is needed. When the import section in a module is adjusted, the module StorageInterfaceReduce will also be added to the list of imported modules enabling invocation of the functions manipulating the storage. The syntax section of this module is given below.

module StorageInterfaceReduce

exports
  context-free syntax

status-is-safe(ATT-NAME, INCINFO)	→ BOOL
status-is-unsafe(ATT-NAME, INCINFO)	→ BOOL
retrieve(ATT-NAME)	→ NORMAL-FORM
store-par-att(ATT-NAME, NORMAL-FORM)	→ BOOL
store-fun-att(ATT-NAME, NORMAL-FORM)	→ NORMAL-FORM

The functions declared in this module will be discussed in detail in Section 7.4. At this point, we only want to show their arguments and result sorts

6.2.2 Transforming the function declarations

The declarations of functions that have been declared as incremental in the syntax sections of a module must be adjusted; the attribute **incremental** must be removed from the declaration. When a module has been transformed into an incremental module, these attributes will not be needed any more. As we described in Section 5.4, the INCINFO must be added as an argument to incremental functions to be able to keep track of the current node in the storage during reduction. For each incremental function declaration, a new declaration of the function is added to the syntax section with this INCINFO argument added.

The incremental functions might be declared in context-free syntax sections that are exported or hidden and it is also possible to have multiple context-free syntax sections within an export and/or hidden section in a module. The function declarations in all sections of a module must thus be transformed. The function declarations with the INCINFO are not all declared in the hidden part of a module. It is possible that these functions are invoked in equations in other incremental modules.

Example 6.1 The function declarations

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

are represented by the following function declarations in the incremental module.

tcs(SERIES,TENV)	→BOOL
tcs(SERIES,TENV,INCINFO)	→BOOL
tcexpr(EXP,TENV)	→TYPE
tcexpr(EXP,TENV,INCINFO)	→TYPE

□

6.2.3 Syntax and semantics of conversion functions

The attributes in the storage we defined in Section 5.2.3, are used for storing and retrieving normal forms that are computed and reused during reduction. We defined the initial NORMAL-FORM in the module Attributes as: |empty|. During the transformation of a module, other NORMAL-FORMs that might occur during reduction of a term must be added.

|SORT| →NORMAL-FORM

We also need functions to *convert* the NORMAL-FORMs back to their original sort. We refer to this type of functions as conversion functions:

nf2SORT(NORMAL-FORM) →SORT

The sorts for which new NORMAL-FORMs and conversion functions are needed are all sorts of terms that might be stored or retrieved during reduction of a term. These sorts can be selected in the declarations of incremental functions. To store and retrieve normal forms in function attributes in the storage, the result sorts of all the incremental functions are needed and to store parameter attributes, the sorts of the parameters in the same functions are needed.

When the sorts have been selected, the declarations of the NORMAL-FORMs and the conversion functions are added to the module. They will only be used within a module and are therefore declared in a context-free syntax section in the hidden part of the module. If there is no hidden section in the module, one will be added.

Example 6.2 For the function declarations

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

the following NORMAL-FORM and conversion functions will be declared in a hidden context-free syntax section of the module:

|BOOL| →NORMAL-FORM
|TYPE| →NORMAL-FORM
|TENV| →NORMAL-FORM
nf2BOOL(NORMAL-FORM) →BOOL
nf2TYPE(NORMAL-FORM) →TYPE
nf2TENV(NORMAL-FORM) →TENV

□

To enable conversion from a NORMAL-FORM to one specific sort, semantics must be defined for the "nf2SORT" functions. Before equations can be defined describing this semantic, we have to be sure that variables are declared for the sorts. It is possible that variables for these sorts are already declared in the module, but these variables could also have been declared in imported modules. We have chosen for a simple solution; for all sorts that are selected for the conversion functions we define new variables. The names of these variables have the form: SORTvar. Perhaps a strange notation for a variable but the probability that variables with the same name have already been declared is small. These variables will only be used in the transformed module and can thus be declared in a hidden part of the module.

Example 6.3 For the definition of the semantics of the conversion functions for the sorts BOOL, TYPE and TENV, the following variables are declared:

BOOLvar → BOOL
TYPEvar → TYPE
TENVvar → TENV

□

The equations defining the conversion functions are very simple. These equations are added to the rest of the equations in the module.

Example 6.4 The equations defining the functions nf2BOOL, nf2TYPE and nf2TENV are given below.

$$\text{nf2BOOL}(|\ \mathit{BOOLvar}\ |) = \mathit{BOOLvar} \quad [\text{Cv1}]$$
$$\text{nf2TENV}(|\ \mathit{TENVvar}\ |) = \mathit{TENVvar} \quad [\text{Cv2}]$$
$$\text{nf2TYPE}(|\ \mathit{TYPEvar}\ |) = \mathit{TYPEvar} \quad [\text{Cv3}]$$

□

6.2.4 Equations introducing the INCINFO

For every incremental function declared in a module, an *indicating* equation is derived. With indicating equation we mean an equation that is only matched when the incremental reduction of a term begins. The purpose of the indicating equations is to initialise the INCINFO. By automatically introducing the INCINFO in an equation, the user does not have to add this information to the term to be reduced. The INCINFO consists of two parts: the rank R and the level L. These two variables are used to keep track of the current node in the storage during reduction with incremental specifications as described in Section 5.4.

For every incremental function declared in the syntax section one equation having this function as its top operator on the left-hand side is selected. The variables representing the constructor on the left-hand side of the equation are substituted by a new variable representing the sort of the constructor. This enables us to derive only one indicating equation per function, instead of one indicating equation per equation defining the function. When the variables have been substituted, a copy is made of this left-hand side and the incremental information, *1:0* (rank R=1 and level L=0), is added to this copy and the copy is made the right-hand side of the equation.

Example 6.5 In the module Pico-typecheck we discussed in Section 2.3.2 there are two equation, [Tc6]-[Tc7], defining the function tcs(SERIES,TENV) →BOOL **{incremental}** and six equations defining the function tcexpr(EXP,TENV) →TYPE **{incremental}**, [Tc11]-[Tc16]. When the indicating equation for these functions are derived only the equations

$$\mathrm{tcs}(\mathit{SERIESvar},\ \mathit{Tenv}) \ = \ \mathrm{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ 1:0) \qquad \text{[AI14]}$$

$$\mathrm{tcexpr}(\mathit{EXPvar},\ \mathit{Tenv}) \ = \ \mathrm{tcexpr}(\mathit{EXPvar},\ \mathit{Tenv},\ 1:0) \qquad \text{[AI16]}$$

are created in the incremental module. □

To be able to define the indicating equations in this way, variables must be declared for all the sorts of the constructors in the module. This is done by selecting the sorts of the constructors in the function declarations of the incremental functions in the module and introducing variables for these sorts in the same way variables were introduced for the conversion functions.

6.2.5 Equations for safe attributes

During incremental reduction the status of an attribute is important. When an incremental function is applied, the status of the related function attribute indicates whether the value stored in the attribute can be retrieved and used as result or whether the term must be re-evaluated. We call an equation *safe* when it specifies the case in which the value stored in an attribute in the storage is reliable and can be retrieved.

These equations are defined in the same way as the indicating equations. Thus we start by selecting the incremental function declarations and continue by adjusting the left-hand side of one arbitrary equation defined for this function in the same way as was done for the indicating equations. The right-hand side and the conditions of the equation are completely changed. The safe equations have one condition. In this condition the function status-is-safe will be invoked during reduction with as argument the name of the function attribute corresponding to the function and the incremental information. The name of the function attribute is derived from the function declaration as described in Section 2.4.2 and the incremental information is the combination R:L, the rank and the level. The new right-hand side of the equation is the invocation of the function retrieve. This function has only one argument: the name of the function attribute which value should be retrieved. The function retrieve returns a term of sort NORMAL-FORM, thus a conversion function is needed to convert this sort into the right sort; the result sort of the function applied.

Example 6.6 The two functions

tcs(SERIES,TENV) $\rightarrow$BOOL **{incremental}**
tcexpr(EXP,TENV) $\rightarrow$TYPE **{incremental}**

in the last example have the following "safe" equations:

$$\frac{\text{true} = \text{status-is-safe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R : L)}{\begin{array}{l}\text{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ R : L) = \\ \text{nf2BOOL}(\text{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS4]}$$

$$\frac{\text{true} = \text{status-is-safe}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ R : L)}{\begin{array}{l}\text{tcexpr}(\mathit{EXPvar},\ \mathit{Tenv},\ R : L) = \\ \text{nf2TYPE}(\text{retrieve}(\texttt{"tcexpr(EXP,TENV)->TYPE"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS6]}$$

The reason why the safe equations are defined as default equations will be discussed in Section 7.4.7. □

6.2.6 Transforming the equations

In this section we describe how the incremental equations in a module are transformed. The incremental equations will be transformed into unsafe equations. Unsafe equations will apply during incremental reduction when the value stored in a function attribute is not reliable. An unsafe equation is bound by two conditions invoking functions defined for the storage: status-is-unsafe and store-fun-att. The function status-is-unsafe is invoked in the first condition during reduction and needs the name of the function attribute to which this equation is related and the incremental information R and L to be able to set the new current node in the storage and determine whether the value of the function attribute is reliable.

When the term on the left-hand side of an equation has parameters, the values of these parameters terms should be stored in the related parameter attributes during reduction by means of the function store-par-att. For each of these parameters a condition is added to the equation in which the function store-par-att is invoked with the name of the related parameter attribute as argument. We described in Section 2.4.2 how the names of parameter attributes are derived.

When conditions have been added to store the values of the parameters, the conditions in the original equation can be transformed. If an incremental function is invoked in these conditions a rank and a level have to be added to these applications. The rank is the child number of the first argument in the function with regard to the constructor of left-hand side of the equation. The level is increased by one: $L + 1$. The variable L was introduced when the left-hand side of the equation was adjusted. Reducing an incremental subterm means stepping one level lower.

The value that should be stored in the function attribute related to the top operator of the transformed equation is the application of the right-hand side of the equation. When the conditions have been adjusted the right-hand side of the equation is transformed. If the right-hand side contains (sub) terms with incremental top operators, a rank and the increased level are added parameters here as well. The complete right-hand side is put between bars, converting its result into an object of sort NORMAL-FORM. The right-hand side between bars is made an argument in the invocation of the function store-fun-att. This function expects two arguments: a normal form and the name of the function attribute related to the function the equation defines. The function store-fun-att returns a NORMAL-FORM thus before this value can be returned as result, this normal form must be converted into the right sort.

Example 6.7 Transforming the equation

$$\frac{\begin{array}{l}\text{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \text{natural}, \\ \text{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \text{natural}\end{array}}{\text{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2,\ \mathit{Tenv}) = \text{natural}} \qquad \text{[Tc14]}$$

into an *unsafe* equation results in the following equation:

$$\frac{\begin{array}{r}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{natural} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad \text{[Tc14]}$$

The rank and level have been added to the left-hand side of the equation, $R : L$. The equation is bound by two new conditions, the first and the fifth. In the first condition the function status-is-unsafe is applied to the function attribute related to the function tcexpr, `"tcexpr(EXP,TENV)->TYPE"` and the incremental information, $R : L$. In the second condition the value of the parameter *Tenv* is stored. This parameter is first converted into an object of sort NORMAL-FORM, $\mid \mathit{Tenv} \mid$. The variable *Tenv* represents the first parameter of the function tcexpr and the related parameter attribute is thus `"par(tcexpr(EXP,TENV)->TYPE,1)"`. In the third and fourth conditions the conditions in the original equation will be evaluated. Both these conditions contains applications of incremental functions. The rank is added to these terms. The variable Exp_1 represents the first child of the constructor on the left-hand side of the equation and the variable Exp_2 represents the second child. The rank in the term in the third conditions is thus set to 1 and in the fourth condition to 2. The level is increased by one, L+1. In the last condition the value of the right-hand side in the original equation is stored by means of the function store-fun-att. This value must first be converted into an object of sort NORMAL-FORM by putting it between bars. The function store-fun-att returns an object of sort NORMAL-FORM but the function tcexpr has return value of sort TYPE. The function nf2TYPE converts the NORMAL-FORM into the sort TYPE.

The equation given below defines an injection as described in Section 5.5.

$$\mathsf{tcs}(\mathit{Stat}, \mathit{Tenv}) \;=\; \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv}) \quad \text{[Tc6]}$$

The function tcs as well as the function tcstat have both been declared as incremental. When equation [Tc6] is transformed, the application of the right-hand side of this equation will be provided with rank 0 to indicate that it defines an injection. The result of transforming this equation is given below.

$$\frac{\begin{array}{r}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, \mid \mathit{Tenv} \mid),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"}, \mid \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv}, 0 : L + 1) \mid)\end{array}}{\mathsf{tcs}(\mathit{Stat}, \mathit{Tenv}, R : L) = \mathsf{nf2BOOL}(\mathit{Nf})} \quad \text{[Tc6]}$$

□

6.2.7 Equations defining the transformation

In this section we present the "top" equations of the generation of an incremental module for the external version of incremental specifications. The function mod2incmod transforms a module into an incremental module and is defined by equation [MIE1] given below. The first argument in the function mod2incmod is an ASF+SDF module in ASFIX representation. The second argument contains the names of other modules that are transformed. In equation [MIE1] information is gathered that is needed when the syntax sections and the equations are transformed. The declarations of functions that have been declared as incremental in the syntax sections of the module are selected by means of the function inc-operators. The values that shall be stored in the storage during reduction must all be converted into the sort NORMAL-FORM. When a value is retrieved from the storage, its sort has to be converted back into its original sort. For this purpose conversion functions are generated and the sorts for which these functions are needed are selected by means of the function conversion-sorts. The sorts of the constructors occurring in the incremental functions are also needed and are selected by means of the functions constructor-sorts.

$$\frac{\begin{array}{l} T_6 = \mathsf{inc\text{-}operators}(\mathsf{cf\text{-}syntax}(T_2)),\\ T_7 = \mathsf{conversion\text{-}sorts}(T_6),\\ T_8 = \mathsf{constructor\text{-}sorts}(T_6),\\ {L_1}' = \mathsf{cc}(\texttt{"Inc"};\ L_1),\\ {T_1}' = [\mathsf{scSourceSyntax\ nil}],\\ {T_2}' = \mathsf{syn2incsyn}(T_2,\ T_5,\ T_7,\ T_8),\\ {T_3}' = [\mathsf{scSourceEquations\ nil}],\\ {T_4}' = \mathsf{eqs2inceqs}(T_4,\ T_7,\ T_6) \end{array}}{\begin{array}{l} \mathsf{mod2incmod}([\mathsf{scModule}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4],\ T_5) =\\ [\mathsf{scModule}\ {L_1}';\ L_2;\ {T_1}';\ {T_2}';\ {T_3}';\ {T_4}'] \end{array}} \quad \text{[MIE1]}$$

To avoid that the original module is replaced by the generated, incremental module, the name of the module is adjusted. The name of the "transformed" module is constructed from the original module name by adding the prefix `Inc` to the module name. The "SourceSyntax" and the "SourceEquations" are normally used to restore comments in specifications but after that a specification has been changed, this is no longer possible and we therefore give these two parts the value nil. The syntax sections and the equations in the module are transformed by means of the functions syn2incsyn and eqs2inceqs that are defined below.

The function syn2incsyn transforms the syntax section of a module. The equation [SIS1] given below defines this function. T in the input represents the syntax of the module in AsFix notation and T_1 is the list of names of other transformed modules. The third argument, T_2 are the sorts for which new normal forms and conversion functions are needed and T_3 is a list of the sorts of the constructors of all incremental functions in the module. The names of other transformed modules in the import section are adjusted by means of the function adjust-import. When the function trans-fun-decls is applied, the function declarations in the syntax section are transformed and for each of these functions declaration a new declaration is created with the INCINFO added to its arguments.

$$\frac{\begin{array}{l} T' = \mathsf{adjust\text{-}import}(T,\ T_1),\\ T'' = \mathsf{trans\text{-}fun\text{-}decls}(T'),\\ T_4 = \mathsf{mk\text{-}new\text{-}normalforms}(T_2),\\ T_5 = \mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T_2),\\ T_6 = \mathsf{mk\text{-}variables}(\mathsf{merge}(T_2,\ T_3)),\\ T_7 = \mathsf{add2imports}(T'',\ [\mathsf{scId}\ \texttt{"StorageInterfaceReduce"}]),\\ T_8 = \mathsf{add2hiddens}(T_7,\ \mathsf{merge}(T_4,\ T_5),\ T_6) \end{array}}{\begin{array}{l} \mathsf{syn2incsyn}([\mathsf{scPrefixSyntax}\ T],\ T_1,\ T_2,\ T_3) =\\ [\mathsf{scPrefixSyntax}\ T_8] \end{array}} \quad \text{[SIS1]}$$

New normal forms are created by means of the function mk-new-normalforms and the "nf2SORT" functions are created by applying the function mk-conversion-fun-decls. New variables for the conversion sorts and the sorts of the constructors in the module are created when mk-variables is invoked.

The module StorageInterfaceReduce is added to the list of imported modules. In this module, the external functions invoked during reduction are defined. The created variables, the declaration of the conversion functions and the definition of the new normal forms are added to a hidden part of the module by applying the function add2hiddens. If no hidden section is present in the module, one will be created.

The equations in a module are transformed when the function eqs2inceqs is applied. By applying the function inc-info-eqs in equation [E2IE] given below, equations are generated in which the INCINFO will be initialised. The function safeeqs generates the *safe* equations. When the function trans-eqs is invoked, the equations are transformed: equations with non-incremental top operators are left intact and the *unsafe* equations are derived from the other equations.

$$\frac{\begin{array}{l} T_3 = \text{inc-info-eqs}(T_2,\ T,\ 1), \\ T_6 = \text{safeeqs}(T_2,\ T,\ 1), \\ T_4 = \text{trans-eqs}(T), \\ T_5 = \text{mk-conversion-eqs}(T_1,\ 1) \end{array}}{\begin{array}{l} \text{eqs2inceqs}([\text{scPrefixEquations } T],\ T_1,\ T_2) = \\ [\text{scPrefixEquations merge}(T_3,\ \text{merge}(T_6,\ \text{merge}(T_4,\ T_5)))] \end{array}} \quad \text{[E2IE]}$$

Finally, in the last condition in equation [E2IE] the semantic for the "nf2SORT" functions is defined.

6.3 Incremental module for ASF+SDF version

Due to the fact that the external functions are not yet present, we have to let the storage be part of the to reduce term. In this section the differences and similarities between the transformation described in the previous section and the generation of an incremental module for the ASF+SDF version will be discussed. The transformation of a module is specified in the module Mod2IncModASDF that can be found in Appendix G.3.1. In Appendix I.2, the result of transforming the Pico-typecheck module into an incremental module for this version is given.

In this transformation the module name is changed in the same way as in Section 6.2.1 and the names of imported modules might also need to be changed. The module StorageInterfaceReduce will not be added to the list of imported modules. In the next chapter we will present the storage management functions that are used before, during and after reduction to manipulate the storage. When the transformed incremental modules are used for the ASF+SDF version, the storage is part of the term to be reduced. This is the way we have specified the storage management functions thus the module SM-during-reduction can be added to the list of imported modules instead. Part of this module is given below.

module SM-during-reduction
imports Propagations$^{H.1}$
exports
  context-free syntax

```
status-is-safe(ATT-NAME, INCINFO, STORAGE)          → BOOL-STORAGE
status-is-unsafe(ATT-NAME, INCINFO, STORAGE)        → BOOL-STORAGE
retrieve(ATT-NAME, STORAGE)                         → NF-STORAGE

store-par-att(ATT-NAME, NORMAL-FORM, STORAGE) → BOOL-STORAGE
store-fun-att(ATT-NAME, NORMAL-FORM, STORAGE) → NF-STORAGE
```

The complete module SM-during-reduction can be found in Appendix H.2. When the storage is part of the to reduce term, it must also be part of the result. For this purpose, two combined sorts have been defined: NF-STORAGE and BOOL-STORAGE. The first is a combination of the sort NORMAL-FORM and sort STORAGE and the second a combination of the sort BOOL and the STORAGE. The functions defined in the module SM-during-reduction will be discussed in detail in the next chapter.

6.3.1 Transforming the function declarations

When the function declaration of an incremental function is transformed its result sort will be changed to the sort NF-STORAGE. The storage must also be added as an argument of the function in the function declaration. An extra function declaration will also be generated with the INCINFO added as in Section 6.2.2.

Example 6.8 The function declarations

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

are represented by the following function declarations in the incremental module:

tcs(SERIES,TENV,STORAGE)	→NF-STORAGE
tcs(SERIES,TENV,INCINFO,STORAGE)	→NF-STORAGE
tcexpr(EXP,TENV,STORAGE)	→NF-STORAGE
tcexpr(EXP,TENV,INCINFO,STORAGE)	→NF-STORAGE

□

6.3.2 New normal forms

Due to the fact that the result sort of the incremental functions have been changed into the sort NF-STORAGE, there is no need to define conversion functions from the sort NORMAL-FORM to a SORT. We must, however, define new normal forms for the result sorts of incremental functions and the sorts of their parameters. These normal forms can be declared in a hidden section of the module.

Example 6.9 For the function declarations

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

the following new normal forms will be declared in a hidden context-free syntax section of the module:

|BOOL| → NORMAL-FORM
|TYPE| → NORMAL-FORM
|TENV| → NORMAL-FORM

□

6.3.3 Equations introducing the INCINFO in the ASF+SDF version

The way equations are derived to generate the *indicating* equations is the same as in the external version. The only difference is that the storage must be passed as an argument and it must also be part of returned normal form.

Example 6.10 For the function declarations

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

the following indicating equations are generated:

$$\mathrm{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ S)\ =\ \mathrm{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ 1:0,\ S) \quad \text{[AII4]}$$

$$\mathrm{tcexpr}(\mathit{EXPvar},\ \mathit{Tenv},\ S)\ =\ \mathrm{tcexpr}(\mathit{EXPvar},\ \mathit{Tenv},\ 1:0,\ S) \quad \text{[AII6]}$$

□

6.3.4 Safe equations

The *safe* equations can be derived in the same way as in Section 6.2.5, however the storage must be added as an argument. The function status-is-safe in the ASF+SDF version returns the combined sort BOOL-STORAGE and the function retrieve yields an object of sort NF-STORAGE.

Example 6.11 For the functions

tcs(SERIES,TENV) →BOOL **{incremental}**
tcexpr(EXP,TENV) →TYPE **{incremental}**

the following "safe" equations are derived.

$$\frac{\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}safe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L, S)}{\begin{array}{l}\mathsf{tcs}(SERIESvar, Tenv, R : L, S) = \\ \mathsf{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"}, S_1)\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS4]}$$

$$\frac{\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}safe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S)}{\begin{array}{l}\mathsf{tcexpr}(EXPvar, Tenv, R : L, S) = \\ \mathsf{retrieve}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, S_1)\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS6]}$$

Note that the function status-is-safe returns a new "copy" of the storage: S_1. When the function status-is-safe is applied, values in this structure are changed. □

6.3.5 Unsafe equations

Of course, the non-incremental equations are left intact as in the previous section when the equations are transformed. The generation of the conditions in which the function status-is-unsafe is invoked and the functions by which the value of the parameters are stored can be accomplished as in Section 6.2.6. Only a minimum of administration is needed to number the variables representing the new copies of the storage. The tricky part in the transformation for the ASF+SDF version is the transformation of the conditions and the parameters in the left-hand side of the equation and the right-hand side of equations that are transformed into *unsafe* equations.

Due to the new result sort of the incremental functions, conditions must occasionally be introduced for terms with incremental top operator occurring in an equation as illustrated in Example 6.12 below.

Example 6.12 The following equation in the Pico-typecheck module is transformed:

$$\mathsf{tcp}(\mathsf{begin}\ Decls\ Series\ \mathsf{end}) \ = \ \mathsf{tcs}(Series, \mathsf{tcd}(Decls)) \qquad \text{[Tc1]}$$

The function tcp, tcs and tcd have all been declared as incremental. The function tcs has one parameter of sort TENV; the result sort of the function tcd in the original specification. In the transformed module all three functions yield objects of sort NF-STORAGE, thus the term tcd(*Decls*) can not be left as a subterm in tcs. A new condition must be generated for the application of tcd:

$|\ TENVvar_i\ |{:}S_{j+1} = \mathsf{tcd}(Decls, S_j)$

The part of the result of applying the function tcd the function tcs needs as argument is $TENVvar_i$. A new condition will also be created for the function tcs, the right-hand side of the equation:

$|\ BOOLvar_k\ |{:}S_{l+1} = \mathsf{tcs}(Series, TENVvar_i, S_l)$

The complete *unsafe* equation corresponding to the equation $[Tc1]$ in the original equation:

$$\frac{\begin{array}{l}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcp(PROGRAM)->BOOL"}, R : L, S),\\ \mid \mathit{TENVvar}_1 \mid : S_2 = \mathsf{tcd}(\mathit{Decls}, 1 : L+1, S_1),\\ \mid \mathit{BOOLvar}_1 \mid : S_3 = \mathsf{tcs}(\mathit{Series}, \mathit{TENVvar}_1, 2 : L+1, S_2)\end{array}}{\begin{array}{l}\mathsf{tcp}(\mathsf{begin}\ \mathit{Decls}\ \mathit{Series}\ \mathsf{end}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcp(PROGRAM)->BOOL"}, \mid \mathit{BOOLvar}_1 \mid, S_3)\end{array}} \quad [\mathsf{ITc1}]$$

A condition has been introduced in which the function status-is-unsafe has been introduced. The variable $\mathit{BOOLvar}_1$ represents the *new* right-hand side of the equation after a condition has been introduced to invoke the function tcs. The ranks 1 and 2 and the increased level $L{+}1$ have been added to the applications of tcd and tcs as in Section 6.2.6. The function store-fun-att in the ASF+SDF version yields an object of sort NF-STORAGE, thus no conversion of this result is needed. □

Example 6.12 illustrate that variables must be declared and introduced when incremental (sub) terms in an unsafe equation are transformed. These variables should be numbered to allow different instantiations of variables declared for the same sort. The sorts for which variables should be defined are the same as the sorts for which we declared new normal forms.

We will name these variables as in the previous section but with the extension that they can be numbered: $\mathit{SORTvar}_i$. The advantage of the way these variables are named is twofold: We do not have to take other variables declared for the same sort into account and the user can easily see which variables have been introduced during the transformation.

Example 6.13 For the sorts BOOL, TYPE and TENV the following variables are declared:

BOOLvar[0-9]⋆ →BOOL
TYPEvar[0-9]⋆ →TYPE
TENVvar[0-9]⋆ →TENV

□

It is possible that a variable defined for a sort has to be used more than once in an unsafe equation. The numbering of these variables has to be maintained. This is done by using a table with these variables and their indices.

It is not necessary to introduce new conditions for all incremental (sub) terms in an equation. Lets say that λ_{inc} is a term with an incremental top operator and $\lambda_{non-inc}$ a term with a non-incremental top operator. We can now distinguish the following cases of conditions:

1. $\lambda'_{inc} = \lambda_{inc}$
2. $\lambda'_{inc} \neq \lambda_{inc}$
3. $\lambda_{inc} = \lambda_{non-inc}$
4. $\lambda_{inc} \neq \lambda_{non-inc}$
5. $\lambda'_{non-inc} = \lambda_{non-inc}$
6. $\lambda'_{non-inc} \neq \lambda_{non-inc}$

In case 1 and 2 above, new conditions must be created for both sides of the conditions:

$$\begin{array}{rcl}\mid \mathit{SORTvar}_i \mid : S_j & = & \lambda'_{inc}\\ \mid \mathit{SORTvar}_{i+1} \mid : S_{j+1} & = & \lambda_{inc}\\ \mathit{SORTvar}_i & \otimes & \mathit{SORTvar}_{i+1} \quad \text{with } \otimes\text{: } = \text{ or } \neq.\end{array}$$

In case 3 no extra condition is needed. The result of transforming this condition will be:

$$| \lambda_{non-inc} | : S_j = \lambda_{inc}$$

If no new condition is introduced for case 4 above, the variable representing the new copy of the storage will not be instantiated, thus:

$$\begin{array}{rcl} | \mathit{SORTvar}_i | : S_j & = & \lambda_{inc} \\ \mathit{SORTvar}_i & \neq & \lambda_{non-inc} \end{array}$$

For case the case 5 and 6 above no conditions need to be created.

Above, we only looked at the top operator of the terms in the conditions. In all six cases for both λ_{inc} and $\lambda_{non-inc}$, new conditions must be created for subterms with incremental operators.

When the term on the right-hand side of an equation is transformed, a new condition must be introduced for the term when it has an incremental top operator. If this term has a non-incremental top operator, conditions will be introduced for its incremental subterms when these are present.

Example 6.14 The following equation is defined in the module Pico-typecheck:

$$\mathsf{tcs}(\mathit{Stat}; \mathit{Series}, \mathit{Tenv}) = \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series}, \mathit{Tenv}) \qquad \text{[Tc7]}$$

The right hand side of this equation has a non-incremental top operator: &. The subterms in this term are the incremental terms: tcstat(*Stat*, *Tenv*) and tcs(*Series*, *Tenv*). For these two terms new conditions must be created.

$$\frac{\begin{array}{r} \mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L, S), \\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, | \mathit{Tenv} |, S_1), \\ | \mathit{BOOLvar}_1 | : S_3 = \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv}, 1 : L + 1, S_2), \\ | \mathit{BOOLvar}_2 | : S_4 = \mathsf{tcs}(\mathit{Series}, \mathit{Tenv}, 2 : L + 1, S_3) \end{array}}{\begin{array}{l} \mathsf{tcs}(\mathit{Stat}; \mathit{Series}, \mathit{Tenv}, R : L, S) = \\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"}, \\ \qquad\qquad | \mathit{BOOLvar}_1\ \&\ \mathit{BOOLvar}_2 |, S_3) \end{array}} \qquad \text{[ITc7]}$$

The terms for which new conditions have been created are thus substituted in the right-hand side of the equation with the variables $\mathit{BOOLvar}_1$ and $\mathit{BOOLvar}_2$. When the subterms have been substituted, a normal form can be converted into the sort NORMAL-FORM, $| \mathit{BOOLvar}_1\ \&\ \mathit{BOOLvar}_2 |$, and can be stored by means of the function store-fun-att. □

6.4 Deriving an InitUpdate module

In this section we will describe how an InitUpdate module can be derived from one or more ASF+SDF modules in ASFIX notation. In this derived module a function init will be declared. This function should be applied when a term in a term editor is changed during editing to initialise or update the storage corresponding to this term. In this section, as in Section 6.2, we assume that the storage can be accessed by means of external functions and is not part of the term reduced. Before we describe the derivation, we show the syntax section of the module StorageInterfaceInit:

module StorageInterfaceInit
exports
  context-free syntax

update-needed({INT ";"}*, INT)	→ BOOL
new-fun-att(ATT-NAME)	→ BOOL
new-par-att({ATT-NAME ";"}+)	→ BOOL
add-deps(DEPS)	→ BOOL

The derived InitUpdate module will import this module. In the equations that are derived for the function init, the functions in the module above will be invoked to manipulate the storage. The functions in this module will be discussed in detail in the next chapter and we only give a short description here.

The function update-needed will be invoked during initialisation or update of the storage to check whether update of a part of the storage is needed. The first argument in this function is a list of the ranks of the children of a node. If a rank in this list has value 0, there will be no node in the storage for this child, see Section 5.3. The functions new-fun-att and new-par-att are invoked when new function or parameter attributes should be created in the storage. The name of a parameter attribute is a list of one or more attribute names. We mentioned in Section 5.2.3 that when parameter attributes are equally defined, only one attribute is created but its name is a list of parameter attribute names. The function add-deps adds dependencies between attributes, thus its argument will be a list of dependencies.

The derivation of an InitUpdate module has been specified in the module DeriveInitUpdateMod that can be found in Appendix G.4.1. For auxiliary modules imported by this module, we refer to the Appendix. The result of deriving an InitUpdate module for the Pico-typecheck module can be found in the Appendix I.3.

6.4.1 Deriving a syntax section

Deriving the syntax for the InitUpdate module is quite simple. The information needed is the names of the modules from which the module is derived and the sorts of the constructors in the incremental functions declared in these modules. The import section in the derived module will contain the names of the modules. During the transformation of the modules, their names are changed thus before the module names can be put in the import section they must be adjusted as well. The prefix `Inc` is added to their names. The module StorageInterfaceInit given above should be imported by the InitUpdate module thus this module name is also added to the list of imported modules.

Given the sorts of the constructors, the function init can be declared. Two declaration for each constructor sort are created: init(SORT) $\rightarrow$ BOOL and init(SORT, INT) $\rightarrow$ BOOL. The INCINFO that is used in the incremental modules is not needed during the updating of the storage. There are no conditions in the equations in this module that can disturb the reduction sequence and we will only be needing the rank to keep track of the current node in the storage.

In equation [IUS] given below the derivation of the syntax section for an InitUpdate module is given. The first argument T_1 is a list containing the sorts of the constructors in the modules for which the InitUpdate module is derived and T_2 is a list containing the names of these modules. Function declarations are created of the form: init(SORT) $\rightarrow$ BOOL when the function init-operators is applied. When the function init-operators-with-rank is applied to the same sorts, function declarations with an extra argument are created. The extra argument is an integer INT, representing a *Rank*: init(SORT,INT) $\rightarrow$ BOOL. Variables are created by means of the function mk-variables. In the derived module, there will also be *indicating* equations. To minimize the number of these equations, new variables are introduced for the constructor sorts. The names of the modules are adjusted when they are applied to the function inc-mod-names. The right-hand side of [IUS] is the AsFix term representing the syntax in the module InitUpdate. The module

`StorageInterfaceInit` is added to the import section.

$$\frac{\begin{array}{l} T_3 = \text{init-operators}(T_2), \\ T_4 = \text{init-operators-with-rank}(T_2), \\ T_5 = \text{mk-variables}(T_2), \\ T_6 = \text{inc-mod-names}(T_1) \end{array}}{\begin{array}{l} \text{init-update-syntax}(T_1, T_2) = \\ [\text{scPrefixSyntax} \\ \ [\text{scImports} \\ \ \ [\text{scId "StorageInterfaceInit"}]; \\ \ \ T_6]; \\ \ [\text{scExports} \\ \ \ [\text{scContextFreeSyntax } T_3]]; \\ \ [\text{scHiddens} \\ \ \ [\text{scContextFreeSyntax } T_4]; \\ \ \ [\text{scVariables } T_5]]] \end{array}} \quad \text{[IUS]}$$

Example 6.15 Below part of the syntax section when an InitUpdate module was derived for the Pico-typecheck module we discussed in Section 2.3.2. This example shows what this section would look like if the only constructor sorts in the module were: PROGRAM and DECLS.

imports StorageInterfaceInit IncPico-typecheck$^{I.1}$
exports
  context-free syntax
    init(PROGRAM) → BOOL
    init(DECLS)   → BOOL
hiddens
  context-free syntax
    init(PROGRAM, INT) → BOOL
    init(DECLS, INT)   → BOOL
  variables
    PROGRAMvar → PROGRAM
    DECLSvar   → DECLS
  □

6.4.2 Equations introducing the rank

Deriving the equations that are “used” to introduce the rank, the indicating equations, is simple when the constructor sorts have been found. An indicating equation is created when the function eqs-intro-rank is applied. Equation [EIR2] given below defines this function.

$$\frac{\begin{array}{l} T_1 = \text{init-operators}([L\ T]), \\ T_2 = \text{init-operators-with-rank}([L\ T]), \\ T_3 = \text{sort2var}(T), \\ T_4 = [\text{scLex "1"}; [\text{scSort scINT}]] \end{array}}{\begin{array}{l} \text{eqs-intro-rank}([L\ T], \mathit{Int}) = \\ [\text{scCondEquation} \\ \ \text{scImplies}; \\ \ \text{cc("IR"; int2lit}(\mathit{Int})); \\ \ [T_1\ \ T_3]; \\ \ [T_2\ \ T_3;\ T_4]; \\ \ [\text{scConditions nil}]] \end{array}} \quad \text{[EIR2]}$$

The input $[L\ T]$ is an ATerm representing a sort. L is the constant scSort and T is a string representing the name of the sort. Two new operators are made for this sort, one with and one without rank, T_2 and T_1. An ATerm representing a variable is made when the function sort2var is applied. In ASF+SDF notation: *SORTvar*. In the fourth condition an ATerm is created representing the lexical 1. The integer *Int* is used to number the tags of the equations. The tag is the concatenation, cc, of `IR` and the *Int* that first has to be converted into a literal.

Example 6.16 For the constructor sorts PROGRAM and DECLS the following indicating equations are derived:

$$\mathsf{init}(\mathit{PROGRAMvar}) = \mathsf{init}(\mathit{PROGRAMvar}, 1) \qquad \text{[IR1]}$$

$$\mathsf{init}(\mathit{DECLSvar}) = \mathsf{init}(\mathit{DECLSvar}, 1) \qquad \text{[IR2]}$$

□

6.4.3 Update (not) needed

When the function init in the derived module is applied to a (sub) term in a term editor, an indicating equation is applied. The next equation to apply will be one defined for a constructor corresponding to the (sub) term in the editor. In the first condition in that equation, the function update-needed will be invoked. This function determines whether the part of the storage corresponding to the term needs to be adjusted and returns a suitable boolean value. When no adjustment is needed, the update procedure should end. We refer to equations defining this case as *update not needed equations.* The other possibility is that the storage needs to be updated and equation defining this case are referred to as *update needed equations.*

For each constructor in modules for which an InitUpdate module is derived, one update not needed equation and one update needed equation must be created. Before these two equations can be created, all equations defining function for the same constructor must be selected.

Example 6.17 When an update not needed equation is created for the constructor represented by begin *Decls*, *Series* end in the module Pico-typecheck, the following equation will be the result:

$$\frac{\mathsf{false} = \mathsf{update\text{-}needed}(1;\,2,\ R)}{\mathsf{init}(\mathsf{begin}\ \mathit{Decls}\ \mathit{Series}\ \mathsf{end},\ R) = \mathsf{true}} \qquad \text{[In1a]}$$

The rank R will be used in the storage to determine which node should be consulted to determine whether update is needed or not. The ranks 1;2 in the first argument of the function update-needed are the ranks of the children of the constructor begin *Decls*, *Series* end. The child represented by the variable *Decls* is the first child and has rank 1 and *Series* is the second child, thus rank 2. □

6.4.4 The update needed equation

When the module InitUpdate is derived, we have little knowledge of the term for which the storage must be initialised. When we derive this module we do not know whether the function ϕ or χ (or $\alpha \ldots \omega$ for that matter) will be applied to an abstract syntax tree T. When the function init in the InitUpdate module is applied to T, the storage must be initialised for all incremental functions defined for T. If the function ϕ and χ are incremental functions defined for a constructor C, attributes related to both ϕ or χ must be added to the storage. The functions ϕ and χ are defined by the following equations:

[1] $\phi(\mathrm{p}(X_1,X_2)) = \psi(X_1)$
[2] $\chi(\mathrm{p}(X_1,X_2)) = \varphi(X_2)$

and that ϕ, χ, ψ and φ have all been declared as incremental. Because incremental functions have been declare for both X_1 and X_2, the storage must be initialised for both these constructor children. It does thus not suffice to look at only one equation. When the storage is initialised, the attributes related to all functions defined for the same constructor must also be added. When the left-hand side of an equation has parameters, the related parameter attributes must also be added. When two parameter are uniquely defined as described in Section 2.4.4, only one parameter attribute needs to be created. Thus we must also compare the definition rules of the parameters.

Dependencies between attributes must also be derived when the InitUpdate module is derived. During the initialisation of the storage, these dependencies must be maintained. An algorithm for deriving dependencies in an equation is given in [Meu94]. We have adopted (and adapted) this algorithm. In the module MakeDepCondEq we have specified the derivation the dependencies in an equations. For details, we refer to Appendix G.4.9.

When an update needed equation is derived, all the aspects mentioned above must be taken into consideration. The algorithm for deriving an update needed equation is as follows:

- Select all equations defined for the same constructor.
- Create a new left-hand side for the new equation with top operator init and as arguments the constructor and the rank *R*.
- Create a list with the ranks of the children of the constructor. The rank in this list representing a child for which no incremental function has been declared is 0.
- Make a condition in which the left-hand side is true and the function update-needed is invoked in the right-hand side. Let the ranks of the children found in the last step be an argument in this function.
- Select the left-hand sides of all equations and make function attributes of their top operators.
- For all of these function attributes, create a condition in which the function new-fun-att is invoked with the function attribute as argument.
- Derive the parameter attributes for the left-hand sides of the equations. Compare the definition rules of all these parameter attributes. If parameter attributes are equally defined, make a list of their names.
- For all parameter attributes found, create a condition in which the function new-par-att is invoked and let the name of the parameter attribute be an argument. For attributes uniquely defined, let the list of their names be the argument.
- For all children of the constructor for which at least one incremental function is defined, create a condition in which the function init is applied to this child. Add the rank of the child as an argument.
- Derive the dependencies for attributes in each equation selected. Make a list of the dependencies and put this list as an argument in a function add-deps. Let the invocation of add-deps be the right-hand side of the created equation.

Example 6.18 In this example, we will describe how an update needed equation is derived from the constructor begin *Decls Series* end when an InitUpdate module is derived for the Pico-typecheck module using the algorithm above. There is only one equation in this module defined for this constructor, [Tc1]:

tcp(begin *Decls Series* end) = tcs(*Series*, tcd(*Decls*)) [Tc1]

The new left-hand side: init(begin *Decls Series* end, *R*). The ranks of the children is 1;2, the functions tcs and tcd are both declared as incremental. The *update needed condition* is thus:

true = update-needed(1;2,*R*). The function attribute related to the the top operator on the left-hand side of the equation, tcp is: `"tcp(PROGRAM)->BOOL"`. The condition in which the function new-fun-att is applied to this function attribute: *Bool* = new-fun-att(`"tcp(PROGRAM)->BOOL"`). The left-hand side of equation [Tc1] has no parameters, thus no conditions for *new parameter attributes* needs to be created. For both *Decls* as well as *Series* conditions must be created in which storage is initialised for these children. The rank of *Decls* is 1 and the rank of *Series* is 2: *Bool* = init(*Decls*, 1) *Bool* = init(*Series*, 2). A dependency has the form *Attribute-name* : *child-nr* → *Attribute-name* : *child-nr*. When an attribute is associated with the right-hand side of the equation, the child number is 0. tcp is defined by the right-hand side of the equation tcs, thus a dependency runs from the function attribute of tcs to the function attribute of tcp: `"tcs(SERIES,TENV)->BOOL"` : 2 → `"tcp(PROGRAM)->BOOL"` : 0. When the dependencies are derived, dependencies between attributes related to subterms can also be derived. The first parameter in tcs is defined by tcd thus a dependency runs from the function attribute of tcd to the parameter attribute related to the first parameter in tcs: `"tcd(DECLS)->TENV"` : 1 → `"par(tcs(SERIES,TENV)->BOOL,1)"` : 2 The combination of found building blocks results in the equation given below.

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(1;\,2,\,R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcp(PROGRAM)->BOOL"}),\\ \mathit{Bool} = \text{init}(\mathit{Decls},\,1),\\ \mathit{Bool} = \text{init}(\mathit{Series},\,2)\end{array}}{\begin{array}{l}\text{init}(\text{begin } \mathit{Decls}\ \mathit{Series} \text{ end},\,R) =\\ \text{add-deps}(\texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \to \texttt{"tcp(PROGRAM)->BOOL"} : 0.\\ \qquad \texttt{"tcd(DECLS)->TENV"} : 1\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2)\end{array}} \quad \text{[In1b]}$$

□

The derivation of an update needed equation has been specified in the module UpdateNeededEq that can be found in Appendix G.4.7.

6.4.5 Superfluous attributes: Who invented injections?

One problem, when the preparations for the initialisation of the storage is done by deriving the InitUpdate module, are the injections. We have not found a proper method to derive the InitUpdate module where the injections do not cause problems. We have however found a technique to repair damage caused. In Section 5.5 we mentioned that attributes related to an injection equation must be lifted. When the equations are selected for a certain constructor also equations defining injections are selected. When equations are selected in the Pico-typecheck module for the constructor represented by *Id* := *Exp*, equation [Tc8] given below will be selected.

$$\text{tcstat}(\mathit{Id} := \mathit{Exp},\,\mathit{Tenv}) \;=\; \text{compatible}(\text{lookup } \mathit{Tenv} \text{ for } \mathit{Id},\,\text{tcexpr}(\mathit{Exp},\,\mathit{Tenv})) \quad \text{[Tc8]}$$

The first argument in the function tcstat is of sort STATEMENT. When the equations are selected, equations defining injections for the sort of this first argument will also be selected. An equation defining an injection for the sort STATEMENT in the Pico-typecheck module is equation [Tc6]:

$$\text{tcs}(\mathit{Stat},\,\mathit{Tenv}) \;=\; \text{tcstat}(\mathit{Stat},\,\mathit{Tenv}) \quad \text{[Tc6]}$$

The result of deriving an update needed equation for these two equations is equation [In6b] given below.

true = update-needed(0; 2, *R*),
Bool = new-fun-att("tcstat(STATEMENT,TENV)->BOOL"),
Bool = new-fun-att("tcs(SERIES,TENV)->BOOL"),
Bool = new-par-att("par(tcstat(STATEMENT,TENV)->BOOL,1)"),
Bool = new-par-att("par(tcs(SERIES,TENV)->BOOL,1)"),
Bool = init(*Exp*, 2)
____ [In6b]
init(*Id* := *Exp*, *R*) =
add-deps("tcexpr(EXP,TENV)->TYPE" : 2
→ "tcstat(STATEMENT,TENV)->BOOL" : 0.
"par(tcstat(STATEMENT,TENV)->BOOL,1)" : 0
→ "par(tcexpr(EXP,TENV)->TYPE,1)" : 2.
"tcstat(STATEMENT,TENV)->BOOL" : 0
→ "tcs(SERIES,TENV)->BOOL" : 0.
"par(tcs(SERIES,TENV)->BOOL,1)" : 0
→ "par(tcstat(STATEMENT,TENV)->BOOL,1)" : 0)

In Figure 6.1 storage tree that has been initialised for the term: x:=5;y:=7. In this figure we use the function names as names of the attributes to save some space. To initialise the term given, another init equation is also needed: [In5b], see Appendix I.3. The attributes that are

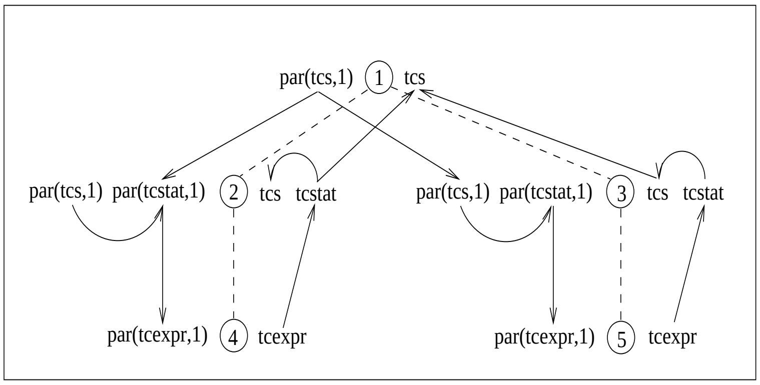


Figure 6.1: The storage tree containing superfluous attributes.

superfluous are the attributes tcs and par(tcs,1) at node number 2. These attributes must be removed. Characteristic for function attributes that are superfluous are that they do not have any successors. The superfluous parameter attribute can be detected as a predecessor of another parameter attribute at the same node that is defined by more than one attribute. In Section 7.3.4 we will explain how these attributes can be removed.

6.5 The ASF+SDF version of the module Init-update

In this section we will describe how an InitUpdate module is derived for the ASF+SDF version. The InitUpdate module for this version will import the module SM-init-update. The functions that are invoked in conditions in the the derived module are defined in this module.

module SM-init-update
imports Dependencies$^{H.4}$
exports
context-free syntax
update-needed({INT ";"}*, INT, STORAGE) → BOOL-STORAGE

$$\begin{array}{ll}
\text{new-fun-att(ATT-NAME, STORAGE)} & \rightarrow \text{STORAGE} \\
\text{new-par-att(\{ATT-NAME ";"\}+, STORAGE)} & \rightarrow \text{STORAGE} \\
\text{add-deps(DEPS, STORAGE)} & \rightarrow \text{STORAGE}
\end{array}$$

These function are defined as the function in the module StorageInterfaceInit we gave in Section 6.4. The differences are the result sorts of these functions. The function update-needed has return sort BOOL-STORAGE, the combination of a boolean value and the storage. The application of the other functions yields a storage.

When an InitUpdate module is derived for the ASF+SDF version, we can use the InitUpdate module derived for the external version. Given this module, it can be transformed into a module for the ASF+SDF version simply by making the storage one of the arguments of the functions defined in this module and by changing the result sorts of the functions. Some variables must also be changed. The derivation of an InitUpdate module for the ASF+SDF version has been specified in the module DeriveInitUpdateASDF, Appendix G.5.1. The result of this derivation of the Pico-typecheck module is given in Appendix I.2.

6.5.1 A new module and its syntax sections

The derivation of an InitUpdate module is simple. The input modules are applied to the function derive-init-update-mod. This function yields an InitUpdate module for the external version. Given this result, it is transformed into an ASF+SDF version by applying the function extc2asdf. These actions are defined in equation [DIUA].

$$\frac{T = \text{derive-init-update-mod}(Ts^{+})}{\begin{array}{l}\text{derive-init-update-ASDF}(Ts^{+}) = \\ \text{extc2asdf}(T)\end{array}} \qquad \text{[DIUA]}$$

In the syntax section, only the list of imported modules and the function declarations must be adjusted. The transformation of the import section is defined in equation [E2A5].

$$\begin{array}{l}\text{extc2asdf([scImports } T_1;\ T_2]) = \\ \quad \text{[scImports [scId "SM-init-update"]; } T_2]\end{array} \qquad \text{[E2A5]}$$

When the import section was created for the external version, the name of the module StorageInterfaceInit was added as the first object in the list of imported modules. In the ASF+SDF version, the module SM-init-update should be imported instead of this module. In equation [E2A5] above, T_1 represents the module StorageInterfaceInit. This module name is replaced by the name of the module SM-init-update.

The function declarations in the syntax sections are adjusted when [E2A9] is applied. T_1 represents the context-free elements of a function: the function name and the list of argument sorts. T_2 is the result sort of the function and T_3 are the attributes attached to the function.

$$\frac{\begin{array}{c} T = \text{storage ( )}, \\ T_1{}' = \text{add2cfelems}(T_1,\ T)\end{array}}{\begin{array}{l}\text{extc2asdf([scCfFunction } T_1;\ T_2;\ T_3]) = \\ \text{[scCfFunction } T_1{}';\ T;\ T_3]\end{array}} \qquad \text{[E2A9]}$$

The function storage() returns a term representing the sort STORAGE in ASFIX notation. The sort STORAGE is added to the list of arguments when add2cfelems is applied. The sort STORAGE is also made the new result sort of the function.

Example 6.19 The following syntax section is the result of transforming the syntax section in Example 6.15.
imports SM-init-update$^{H.5}$ IncPico-typecheck$^{I.1}$

exports
 context-free syntax
  init(PROGRAM, STORAGE) → STORAGE
  init(DECLS, STORAGE)   → STORAGE
hiddens
 context-free syntax
  init(PROGRAM, INT, STORAGE) → STORAGE
  init(DECLS, INT, STORAGE)   → STORAGE
 variables
  PROGRAMvar → PROGRAM
  DECLSvar   → DECLS
 □

6.5.2 Number the variable S

In the equations in the InitUpdate module the variable S represents the storage. During initialisation of the storage, several copies of the storage will be returned thus this variable must be numbered. On the right-hand side of the conditions functions are applied to the storage and the same functions return (new) copies of the storage. Thus when a function is applied to the storage, represented by the variable S_{Index}, the index of the variable on the other side of the conditions should be increased: $S_{Index+1} = \mathsf{fun}(S_{Index})$ When a condition is transformed, its the right-hand side is transformed, the *Index* is increased by one and its left-hand side is transformed. The transformation of a condition is defined by equation [E2AI2].

$$\frac{\begin{array}{r}T_2{}' = \mathsf{extc2asdf}(T_2,\ \mathit{Index}),\\ \mathit{Index}_1 = \mathsf{increase}(\mathit{Index}),\\ T_1{}' = \mathsf{extc2asdf}(T_1,\ \mathit{Index}_1)\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([\mathsf{scCondition}\ L;\ T_1;\ T_2],\ \mathit{Index}) =\\ \mathit{Index}_1\ /\ [\mathsf{scCondition}\ L;\ T_1{}';\ T_2{}']\end{array}} \qquad \text{[E2AI2]}$$

The right-hand side of the condition is represented by T_2 and the left-hand side by T_1. The new index Index_1 is parts of the result of the transformation; when the next condition is transformed, this new index should be used.

The term on the left- and right-hand side of a condition is either a variable or a function application. Equation [E2AI3] applies when this term is a variable.

$$\mathsf{extc2asdf}([\mathsf{scVar}\ T],\ \mathit{Index})\ =\ \mathsf{storage}(\mathit{Index}) \qquad \text{[E2AI3]}$$

When the term is a variable, this variable is replaced by S_{Index}. The function storage(*Index*) returns an AsFixTerm representing S_{Index}. If the term is a function application, equation [E2AI4] applies.

$$\frac{\begin{array}{r}T' = \mathsf{extc2asdf}([\mathsf{scCfFunction}\ T]),\\ T_1{}' = \mathsf{merge}(T_1,\ \mathsf{storage}(\mathit{Index}))\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([[\mathsf{scCfFunction}\ T]\ T_1],\ \mathit{Index}) =\\ [T'\ T_1{}']\end{array}} \qquad \text{[E2AI4]}$$

In the first condition the function declaration ([scCfFunction T]) of the term is adjusted. When the function extc2asdf is applied to this function declaration, equation [E2A9] given in Section 6.5.1 will apply. The variable S_{Index} is added to the list of arguments, T_1.

6.5.3 The update needed condition

The only exception that has to be made during this transformation concerns the first condition in an equation. If an equation in the InitUpdate module has conditions, its first condition will contain the invocation of the function update-needed. This condition is an exception because the result sort of this function is BOOL-STORAGE and not STORAGE as the rest of the functions. When the conditions in an equation are transformed, the first condition must be applied to the function extc2asdf-cond defined by equation [E2AC2] given below.

$$\frac{\begin{array}{r@{\;}l} T_2{}' &= \text{add-s-rhs-un}(T_2,\ \mathit{Index}), \\ \mathit{Index}_1 &= \text{increase}(\mathit{Index}), \\ T_1{}' &= \text{add-s-lhs-un}(T_1,\ \mathit{Index}_1) \end{array}}{\begin{array}{l} \text{extc2asdf-cond}([\text{scCondition } L;\ T_1;\ T_2],\ \mathit{Index}) = \\ \mathit{Index}_1\ /\ [\text{scCondition } L;\ T_1{}';\ T_2{}'] \end{array}} \qquad \text{[E2AC2]}$$

The right-hand side of the condition is applied to the function add-s-rhs-un that adjusts the term update-needed(*Ranks*,*R*), the index used to number the storage is increased and the left-hand side of the condition is transformed by means of the function add-s-lhs-un. This left-hand side is always a boolean constant: true or false. The increased index is part of the result yielded by the function extc2asdf-cond.

The function add-s-rhs-un transforms the term update-needed(*Ranks*,*R*) by adding the variable *S* with the proper index as last argument: update-needed($Ranks$,R,S_{Index}) The equation [ASrhs] given below defines this function.

$$\frac{\begin{array}{r@{\;}l} T_1{}' &= \text{add2cfelems}(T_1,\ \text{storage ( )}), \\ T_2{}' &= [\text{scSort scBOOL-STORAGE}], \\ T_4{}' &= \text{merge}(T_4,\ \text{storage}(\mathit{Index})) \end{array}}{\begin{array}{l} \text{add-s-rhs-un}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4],\ \mathit{Index}) = \\ [[\text{scCfFunction } T_1{}';\ T_2{}';\ T_3]\ T_4{}'] \end{array}} \qquad \text{[ASrhs]}$$

The sort STORAGE as added to the list of argument sorts by means of the function add2cfelems. The function is given the new result sort BOOL-STORAGE and the variable S_{Index} is added to the list of arguments in the term. The function storage(*Index*) returns an AsFixTerm representing the variable S_{Index}.

The left-hand side of an *update needed* condition is always a boolean constant: true or false. This constant must be combined with a variable representing the storage: true : S_{Index} or false : S_{Index}. This combined sort is created when the function add-s-rhs-un defined by equation [ASrhs] is applied to a boolean constant and a proper index for the variable *S*.

add-s-lhs-un(*T*, *Index*) = [ASlhs]

```
[[scCfFunction
  [scCfElems [scSort scBOOL]; [scQLit ":"]; storage ( )];
  [scSort scBOOL-STORAGE];
  [scAttributes nil]]
 T; storage(Index)]
```

6.5.4 Transforming an equation

The transformation of an equation is defined by equation [E2A10].

$$\frac{\begin{array}{r@{\;}l} T_1{}' &= \text{extc2asdf}(T_1,\ \texttt{"1"}), \\ \mathit{Index}\ /\ T_3{}' &= \text{extc2asdf-cond}(T_3,\ \texttt{"1"}), \\ T_2{}' &= \text{extc2asdf}(T_2,\ \mathit{Index}) \end{array}}{\begin{array}{l} \text{extc2asdf}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ [L\ T_3]]) = \\ [\text{scCondEquation } L_1;\ L_2;\ T_1{}';\ T_2{}';\ [L\ T_3{}']] \end{array}} \qquad \text{[E2A10]}$$

The left-hand side of the equation, T_1 is transformed; the variable S_1 will be added to its list of arguments. The list of conditions, T_3, is applied to the function extc2asdf-cond. The first condition in this list is the update needed condition and will get the special treatment. The rest of the conditions are transformed as described in Section 6.5.2. The transformation of the conditions yields a new index, *Index*. The variable S_{Index} can be added to the right-hand side of the equation, T_2.

Example 6.20 Equation [IR1] given below is the result of transforming equation [IR1] in Example 6.16:

$$\mathsf{init}(\mathit{PROGRAMvar}, S_1) = \mathsf{init}(\mathit{PROGRAMvar}, 1, S_1) \qquad \text{[IR1]}$$

When equation [In1b] in Example 6.18 is transformed, results in the following equation:

$$\frac{\begin{array}{l} \mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1), \\ \qquad\; S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcp(PROGRAM)->BOOL"}, S_2), \\ \qquad\; S_4 = \mathsf{init}(\mathit{Decls}, 1, S_3), \\ \qquad\; S_5 = \mathsf{init}(\mathit{Series}, 2, S_4) \end{array}}{\begin{array}{l} \mathsf{init}(\mathsf{begin}\ \mathit{Decls}\ \mathit{Series}\ \mathsf{end}, R, S_1) = \\ \mathsf{add\text{-}deps}(\texttt{"tcs(SERIES,TENV)->BOOL"} : 2 \\ \qquad \rightarrow \texttt{"tcp(PROGRAM)->BOOL"} : 0. \\ \qquad \texttt{"tcd(DECLS)->TENV"} : 1 \\ \qquad \rightarrow \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2, S_5) \end{array}} \qquad \text{[In1b]}$$

□

6.6 Deriving the PathSearch module

In this section we will describe how a PathSearch module can be derived. In this module a function path will be defined. This function compares two abstract syntax trees (terms) and returns a path to the top of the first (sub) tree where the two trees are not equal. A path is a list of *ranks*, child numbers, that can be followed from the top of a tree to find a certain node in the tree. We define the rank of the top of a tree as 1.

Example 6.21 In Figure 6.2 the path to the node in which the two trees differ is 1;1;2. All subterms in the two trees are equal except `x` and `x'`. □

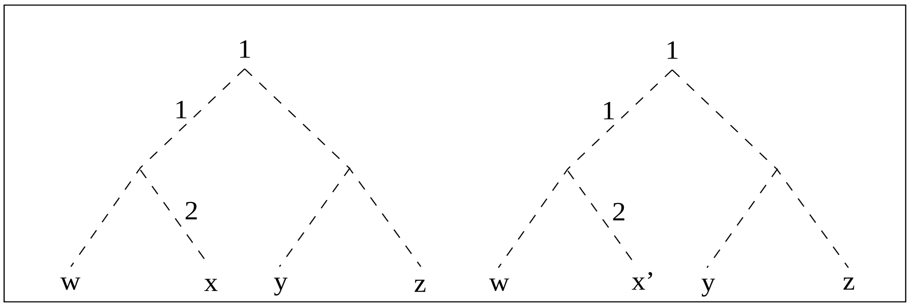


Figure 6.2: Two abstract syntax trees with the ranks indicating the path to the node in which they differ.

If the storage used during incremental reduction has been initialised for a term in a term editor and this term is changed, the storage must be updated; the storage must correspond to the new version of the term in the editor before that term can be reduced incrementally. The function path will be defined to compare 2 terms: the old term for which the storage has been initialised and the new term. Because there is a one-to-one mapping between the nodes in the storage tree and the nodes in the abstract syntax tree representing the term, the found path can be used to locate the part of the storage that needs to be updated. When two terms are compared, we distinguish the following tree cases:

- old-term = new-term

- the old-term and the new-term differ in exactly one direct subterm.
- the old-term and the new-term differ in more than one direct subterm.

In the first case above, we define the path as 0 indicating that the two terms are equal. In the second case, the two (direct) subterms that differ should be compared, and in the last case we define the two terms as completely different and the path is *empty*.

To be able to create the path, the rank of the (sub) terms is needed. When two terms are compared the rank of those terms must be available. The following equations are defined to find a path:

[1] path(*old-term*, *new-term*) = 0
 when *old-term* = *new-term*
[2] path(*old-term*, *new-term*) = empty
 when the terms differ in more than one direct subterm
[3] path(*old-term*, *new-term*) = path(*old-term*, *new-term*, 1)
 when the terms differ in exactly one direct subterm
[4] path(*old-term*, *new-term*, *R*) = R
 when *old-term* = *new-term*
[5] path(*old-term*, *new-term*, *R*) = *R*; path($subterm_{old}$, $subterm_{new}$, *R*')
 when $subterm_{old} \neq subterm_{new}$ and no other direct subterms differ.
 R' is the rank of the subterms.
[6] path(*old-term*, *new-term*,*R*) = R
 when the terms differ in more than one direct subterm

Equation 1-3 above apply when the comparison of two terms begins. In equation 3 the rank is introduced; we defined the rank of the top of a tree as 1. If we would derive equations such as equation 1-6 above, a lot of equations would be generated; one for each direct subterm in a term. We have chosen for another approach. The derivation of a PathSearch module has been specified in the module DerivePathSearchMod that can be found in Appendix G.6.1. The result of generating a PathSearch module for the Pico-typecheck module can be found in Appendix I.5.

6.6.1 The auxiliary path search module

The generated module PathSearch imports the following module:

```
module AuxPathSearch
imports InitUpdate^I.3 Integers^A.3
exports
  sorts PATH PATHS TERM
  context-free syntax
    {INT ";"}*                                   → PATH
    {PATH ":"}*                                  → PATHS
                                                 → TERM
    term-top                                     → INT
    "path(" TERM "AND" TERM ")"                  → PATH
    "path(" TERM "AND" TERM "," INT ")"          → PATH
    comp-paths({PATH ":"}*)                      → PATH
    rm-term-top(PATH)                            → PATH
  variables
    Term [0-9']* → TERM
    P [0-9']*    → {INT ";"}*
    Ps [0-9]*    → {PATH ":"}*
    R [0-9']*    → INT
```

In the module above the sort TERM is defined as the empty sort. During the generation of the

PathSearch module we will define all the sorts that might be compared during the path search as TERMs. Below we show the definition of the function path. These two equations will apply when the comparison of two terms begins. Equation [BPATH2] is used to introduce the rank of the top terms. The reason why the constant term-top is introduced and the application of the function rm-term-top will be explained below.

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}) \ = \ 0 \qquad \text{[BPATH1]}$$

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}') \ = \ \text{rm-term-top}(\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}', \text{term-top})) \ \ \textbf{when}\ \ \mathit{Term} \neq \mathit{Term}' \qquad \text{[BPATH2]}$$

For the path search with the rank introduced, we have defined the following equations in this module:

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}, R) \ = \ 0 \qquad \text{[PATH1]}$$

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}', \text{term-top}) \ = \quad \textbf{otherwise} \qquad \text{[PATH]}$$

$$\frac{R \neq \text{term-top}}{\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}',\ R) = R} \quad \textbf{otherwise} \qquad \text{[PATH]}$$

Equation [PATH1] applies when two terms are equal and the default equations will apply when the two terms have more than one direct subterms that are different or are completely different. The first default equation will apply when the rank of the terms being compared is term-top and covers the case when the old-term and the new-term are completely different and the emtpy path should be returned. The second default equation will apply when more than one subterm of the old-term and the new-term are different. In this case the rank must be returned.

When the second default equation above has not been applied during the search for the path, the constant term-top should be replaced by the rank 1, the rank of the top of the term. This is done when the function rm-term-top defined below is applied.

$$\text{rm-term-top}(\text{term-top};\ P) \ = \ 1;\ P \qquad \text{[RM-top1]}$$

$$\text{rm-term-top}(P) \ = \ P \quad \textbf{otherwise} \qquad \text{[RM-top]}$$

During the generation of the PathSearch module, equations will be introduced for the case when exactly one direct subterm differ.

Example 6.22 The following equation is part of the Pico-typecheck specification (see Section 2.3.2):

$$\frac{\begin{array}{c}\text{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \text{natural},\\ \text{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \text{natural}\end{array}}{\text{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2,\ \mathit{Tenv}) = \text{natural}} \qquad \text{[Tc14]}$$

The function tcexpr has been declared as incremental. When an equation corresponding to the equation above is defined for the function path, the result will be:

$$\frac{\begin{array}{l}P_1 = \text{path}(\ \mathit{Exp}_1\ \text{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \text{path}(\ \mathit{Exp}_2\ \text{AND}\ \mathit{EXPvar}_2,\ 2),\\ P_3 = \text{comp-paths}(P_1\!:\ P_2),\\ P_3 \neq 0\end{array}}{\text{path}(\ \mathit{Exp}_1 + \mathit{Exp}_2\ \text{AND}\ \mathit{EXPvar}_1 + \mathit{EXPvar}_2,\ R) = R;\ P_3} \qquad \text{[PS8]}$$

In the first condition the first direct subterms represented by the variables Exp_1 and EXPvar_1 will be compared and a path P_1 will be found. The rank of these subterms is 1 and is given as an argument. In the second condition the path P_2 is computed by applying the function path to the second direct subterms and their rank 2. The two paths found are compared by means of the function comp-paths that yields a new path. If this path, P_3, is not equal to 0, P_3 is combined with the rank R and returned as the path found. When the path P_3 is 0 the rank of the compared terms R, should not be returned. In this case, the fourth condition forces the application of equation [PATH1] above. The definition of the function comp-paths is given below. □

Below the equations defining the function comp-paths is given.

$$\mathsf{comp\text{-}paths}(Ps_1\colon 0\colon Ps_2\colon P\colon Ps_3) = \mathsf{comp\text{-}paths}(Ps_1\colon Ps_2\colon P\colon Ps_3) \quad \text{[CompPs1]}$$
$$\mathsf{comp\text{-}paths}(Ps_1\colon P\colon Ps_2\colon 0\colon Ps_3) = \mathsf{comp\text{-}paths}(Ps_1\colon P\colon Ps_2\colon Ps_3) \quad \text{[CompPs2]}$$
$$\mathsf{comp\text{-}paths}(P) = P \quad \textbf{when } P \neq 0 \quad \text{[CompPs3]}$$
$$\mathsf{comp\text{-}paths}(0) = 0 \quad \text{[CompPs4]}$$
$$\mathsf{comp\text{-}paths}(Ps) = \quad \textbf{otherwise} \quad \text{[CompPs]}$$

In equation [CompPs1] and [CompPs2] occurrences of the path 0 are removed until there are only paths unequal to the path 0 left. If there are no paths unequal to 0, one path 0 will be left. Equation [CompPs2] applies when there is exactly one path P. The default applies when the function comp-paths has been applied with more than one path that are not equal to 0. When this occurs, more than one direct subterms were not equal thus the empty path must be returned.

6.6.2 New variables

As you might have noticed in Example 6.22 that new variables are used in the generated equations as in the generated modules discussed in the previous sections in this chapter. The sorts for which variables must de declared in the PathSearch module are not easy to select; they must be found in the equations in the modules for which the PathSearch module is generated. We start by selecting all equations having incremental top operators on their left-hand sides. The sorts of the variables in the first argument on the left-hand side of these equations are selected, except left-hand sides in which the first argument is a single variable.

Example 6.23 The functions tcd, tcidlist and tcstat have all been declared as incremental and are defined by the following equations.

$$\mathsf{tcd}(\mathsf{declare}\ \mathit{Id\text{-}type\text{-}list};) = \mathsf{tcidlist}(\mathit{Id\text{-}type\text{-}list}, \mathsf{empty\text{-}tenv}) \quad \text{[Tc2]}$$

$$\mathsf{tcidlist}(\mathit{Id\text{-}type}, \mathit{Tenv}) = \mathsf{tcid}(\mathit{Id\text{-}type}) + \mathit{Tenv} \quad \text{[Tc3]}$$

$$\mathsf{tcstat}(\mathit{Id} := \mathit{Exp}, \mathit{Tenv}) = \mathsf{compatible}(\mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id}, \mathsf{tcexpr}(\mathit{Exp}, \mathit{Tenv})) \quad \text{[Tc8]}$$

The sort of the variable *Id-type-list* in equation [Tc2] will be selected for the declaration of new variables. It is embedded in the expression: declare *Id-type-list*;. The sort of the variable *Id-type* in equation [Tc3] will not be selected because the first argument is a single variable. In equation [Tc8] the sorts of the variables *Id* and *Exp* will be selected. □

The reason why we do not select sorts of single variables such as *Id-type* in Example 6.23 is that no equations defining the function path will be generated for equations in which only a single variable occur. If this kind of equations would be generated, they would introduce infinite loops: path(*Term*, *Term'*) = path(*Term*, *Term'*). We need to give the variables indices during the generation of the equations defining the function path, thus the variables will be declared with this option.

6.6.3 Define new TERMs: avoid ambiguities

We will now explain how the sorts can be selected that should be injected into the sort TERM. By defining these injections we can use the equations defining the function path in the module AuxPathSearch. When these injections have been defined, we only need to define one equation for each distinct constructor (the first argument) occurring in the left-hand side of the equations defining incremental functions. When these sorts are selected, we must be careful to avoid ambiguities.

We start by selecting the sorts of the constructors in functions declared as incremental: the C-sorts. The sorts selected for the declarations of the variables are called the V-sorts. The last sorts we need to determine what sorts should be injected are the sorts of injections that have been defined in the specification for which the PathSearch module is generated. The injections we need to find can easily be detected at AsFix level.

Example 6.24 The sort of the first argument in the function declaration tcxepr(EXP, TENV) → TYPE is the sort EXP and is defined by the following equation:

tcexpr(*Id*, *Tenv*) = lookup *Tenv* for *Id* [Tc11]

In equation [Tc11] the sort of the variable *Id* is ID, see Appendix B.1. Thus somewhere in the specification the injection ID → EXP has been defined. In Example 6.23 the sort ID was selected as a variable sort. If we would inject all variable sorts we found in that example in the sort TERM, the sort ID would end up with the definition: ID → EXP and ID → TERM. □

Once the injections have been found, all sorts $Sort_1$ in the result of the union of the V-sorts and the C-sorts are removed when an injection $Sort_1 \rightarrow Sort_2$ exist and $Sort_2$ is also present in the union. When the injection sorts have been removed the set of sorts that should be injected has been found.

Example 6.25 In the Pico-typecheck module the following variable sorts would be selected: V-sorts:{DECLS, SERIES, ID-TYPE-LIST, STATEMENT, ID, EXP}. The sorts of the constructors: C-sorts:{PROGRAM, DECLS, ID-TYPE-LIST, SERIES, STATEMENT, EXP}. The union of these two sets results in the following set: {PROGRAM, DECLS, ID-TYPE-LIST, SERIES, STATEMENT, EXP, ID}. In the equations in this module, the following injections can be found: {ID-TYPE → ID-TYPE-LIST, STATEMENT → SERIES, ID → EXP, NAT-CON → EXP, STR-CON → EXP}. When the injection sorts have been remove, we can define the following injections :

```
hiddens
  context-free syntax
    PROGRAM      → TERM
    DECLS        → TERM
    ID-TYPE-LIST → TERM
    SERIES       → TERM
    EXP          → TERM
```

□

6.6.4 Generation of equations

We start the generation of the equations by selecting all distinct constructors in the left-hand sides of equations defining incremental functions. The constructors that are represented by a single variable are excluded. For each of these constructors a new equation defining the function path is generated. We start by making a copy of the constructor and in this copy the variable names are changed into variables having the form: $SORTvar_i$. The left-hand side of the new equation is created by letting the original constructor be the first argument and its copy with the changed variable names the argument. The variable R is added as the third argument.

The conditions in which the paths of the subterms should be computed can be generated by using the lists of variables in the constructor, $V_1 \ldots V_n$ and the list of variables in its copy, $V'_1 \ldots V'_n$.

For all variables i in these list, a condition is created: P_i = *path(*V_i AND V_i', i). The last two conditions in the equation are: P_{n+1} = *comp-paths(*P_1 : $\cdots$:P_n') and $P_{n+1} \neq 0$. Thus once the constructors have been found, the generation of the equation is straightforward.

6.7 Conclusions

We have in ASF+SDF specified transformations and derivations of ASF+SDF modules in ASFIX notation. The first transformation transforms a module into an incremental module. The generated module can be used to reduce terms incrementally. The second transformation also generates an incremental module. The third derives an InitUpdate module that should be used during editing to initialise and update the storage. The fourth also derives an InitUpdate module. The result of the first and the third transformation can be used when the use of external functions is supported. The results of the other two generations can be used when the storage is part of the term that is reduced incrementally. The fifth module that can be generated is the PathSearch module that should be used to determine what parts of an term in a term editor have been changed. With the information provided by this module, the right parts of the storage can be adjusted.

When a term is reduced, using the incremental module, 3 steps will always be taken when the attribute corresponding to the top operator of that term is reliable. If this attribute is not reliable, the number of steps that should be taken during non-incremental reduction will be taken plus 4 extra steps for checking the status of the attribute and storing the new normal form. If the term has parameters, 2 extra steps are needed to store the normal form of each parameter in its corresponding parameter attribute.

Chapter 7

Storage before, during and after reduction

7.1 Introduction

During incremental reduction with incremental specifications normal forms are stored in attributes in the storage we defined in Chapter 5. The status of these attributes is important to assure proper and effective reductions; (sub) terms should only be re-computed when necessary. In Chapter 6 we presented a method for transforming modules into incremental modules and how an InitUpdate module is derived from modules. During these processes conditions are added to the equations in which functions are invoked to initialisation and updating the storage during editing and the updating and use the storage during reduction. We refer to these functions as "storage management" functions to which this chapter is devoted. We will, however, start in Section 7.2 with discussing when and why the attributes should be updated and briefly discuss the functions resetting the attribute status: the attribute propagations. In Section 7.3 the storage management functions used to initialise and update the storage during editing are introduced followed by a presentation of the storage management functions used during reduction in Section 7.4. The discussion of the storage management functions is completed in Section 7.5 where the storage management function that should be applied after reduction is introduced.

7.2 Updating the attributes

Since an attribute value depends on other attribute values, the status of one attribute may be important for its successors. We start with a description of the status an attribute can have. The functions that reset the status of attributes and their predecessors and successors are referred to as *attribute propagation functions*. These functions are invoked frequently before, during and after incremental reduction, thus a brief description of each of them is given followed by a discussion concerning when and why the status of the attributes should be reset.

7.2.1 Status indications

We will now give a description of the differences between the several status indications. Both status C (Changed) and status UC (UnChanged) indicates that an attribute value is reliable and we call both the attribute and its value safe. During reduction, attributes obtain status C (Changed) when they have obtained a new value. Status UC (UnChanged) is used for attributes that have a correct value that has not been changed during reduction.

The status IN (INitial) and TE (TobeEvaluate) is given to attributes of which the value is not necessarily correct, and we refer to such attributes and their value as unsafe. The difference between IN (INitial) and TE (TobeEvaluate) is that the value of a "TobeEvaluated" attribute is consistent with the value of its predecessors with status UC (UnChanged), IN (INitial) and TE (TobeEvaluate). This need not be the case for "Initial" attributes. When during reduction the values of the predecessors of a "TobeEvaluated" attribute remain the same, this attribute need not be re-evaluated and its status can be reset to UC (UnChanged). This in contrast to attributes with status IN (INitial), for which the value must always be re-computed.

7.2.2 Attribute propagations

The algorithms for the attribute propagations are given in [Meu94]. We have adopted these algorithms and specified the propagation functions in the module Propagations that can be found in Appendix H.1. There are four propagation functions: prop-te-up, prop-te-down, prop-unchanged and reset-changed.

The function prop-te-up

The function prop-te-up is applied to function attributes only. For all direct successors of this attribute with status C (Changed) or UC (UnChanged) the status is set to TE (TobeEvaluated). The function prop-te-up is applied to this successor if the successor is a function attribute. If the successor is a parameter attribute, the status of its corresponding functions attributes is set to TE (TobeEvaluated) and the function prop-te-up is invoked with these function attributes. When the function prop-te-up has been applied to a function attribute, all its *horizontal* and *upward* successors have status TE (TobeEvaluated) or IN (INitial).

Example 7.1 The second storage-tree in Figure 7.1 shows the effect of the function prop-te-up when it has been applied to the function attribute at node N_2. □

The function prop-te-down

The function prop-te-down is applied to parameter attributes only. For all direct successors of this attribute that are parameters attributes and have status C (Changed) or UC (UnChanged) the status is set to TE (TobeEvaluated) and for all the corresponding function attributes of these successors, the function prop-te-up is applied after that their status has been set to TE (TobeEvaluated).

Example 7.2 The third storage-tree in Figure 7.1 shows the effect of the function prop-te-down when it has been applied to the parameter attribute at node N_3 in the storage. □

The function prop-unchanged

The function prop-unchanged can be applied to both function attributes as well as parameter attributes that have status unchanged. For all direct successors of the attribute with status TE (TobeEvaluated) the status is set to UC (UnChanged) and the function prop-unchanged is applied to this successor when all the predecessors of the successor have status UC (UnChanged).

The function prop-unchanged is the reason why we need two status indications for attributes with a safe value and two for attributes with an unsafe value: Only "TobeEvaluated" attributes are reset to "UnChanged", not "Initial" attributes. The status of a "TobeEvaluated" attribute is reset only if all its predecessors have status UC (UnChanged). Successors of a "Changed" attribute must always be re-evaluated.

The function reset-changed

The function reset-changed should be applied to an attribute with status C (Changed). Its status is set to UN (UnChanged) and all its successors (and corresponding function attributes if it is a parameter attribute) with status TE (TobeEvaluated) are given status IN (INitial).

Example 7.3 If the function reset-changed should be applied to the last storage tree in Figure 7.1, the status of all attributes would be set to UC (UnChanged). □

7.2.3 Updating the attributes

In this section, we explain when and why the status of the attributes should be updated. Initially, before a function $\phi \in \Phi$ can be applied to an edit term T the storage must be initialised. This is done by applying the function init [1] to T. The first time the storage is initialised for a term T, all attributes that are created in the storage, have status IN (INitial). During reduction of $\phi(T)$, the attribute ϕ of T in the storage as well as all its predecessors in the storage obtain a (new) value. When the value in an attribute is changed, its status is set to C (Changed). After reduction all "Changed" attributes are reset to "UnChanged".

After this initial reduction the status of the attributes in the storage is updated at three stages: during (re) initialisation of the storage, during reduction when a (new) value is stored in an attribute, and after reduction to reset the status of all "Changed" attributes to "UnChanged".

Attribute updating during init/update storage

Assume that T_{ast} is an abstract syntax tree of a program in an editor and T_s is the storage corresponding to T_{ast} and that the attributes in T_s have obtained a value when a function in an incremental specification was applied to T_{ast}.

After modifying a part of the text in the editor a path must be found in T_{ast} to the subtree T'_{ast} which represents the modified part in T_{ast}. Given this path, we will be able to find the node N_s in the storage that corresponds to the top of T'_{ast}. All the children of the node N_s will be removed and the a new subtree T'_s will be initialised in such a manner that it corresponds to the subtree T'_{ast}. The node N_s will be the top of this new subtree T'_s in the storage. The attributes in T'_s all get the status IN (INitial). The attributes at N_s still contain the value they were given during the last reduction. They are no longer safe and their value is no longer consistent with the value of their (new) predecessors. Their status is therefore set to IN (INitial).

As a result of the initialisation of the subtree T'_s, the value of attributes that depend on the function attributes at the node N_s that is the top of T'_s, has become unsafe. We do not mark all these successor attributes as unsafe before reduction starts. It suffices to do so only for those that are on a path of horizontal and upward dependencies from a function attribute at N_s to an attribute at the top of the storage, because other affected attributes can be marked during reduction. After the function prop-te-up we described in Section 7.2.2 has been applied to a function attribute at the node N_s, all its *horizontal* and *upward* successors have status TE (TobeEvaluate). If the storage is initialised more that once between reductions, these successors can also have status IN (INitial).

Example 7.4 The second storage-tree in Figure 7.1 shows the effect of updating the storage after the corresponding abstract syntax tree T_{ast} has been changed. □

Attribute updating during reduction

When a term $\phi(T')$ is applied to the equations in the incremental specification, the functions store-par-att and store-fun-att are applied when a normal form must be stored in a parameter attribute or a function attribute. The new value is compared with the old value, if they are different the status is set to "Changed", otherwise is becomes "UnChanged". A "Changed" parameter attribute at the node N_s can have successors attributes at the children of N_s that are not yet unsafe. After the function prop-te-down has been applied to these "Changed" parameter attributes, all direct and transitive successors at the children of N_s are marked as "TobeEvaluated" of "Initial".

[1] This function will be discussed in the next section.

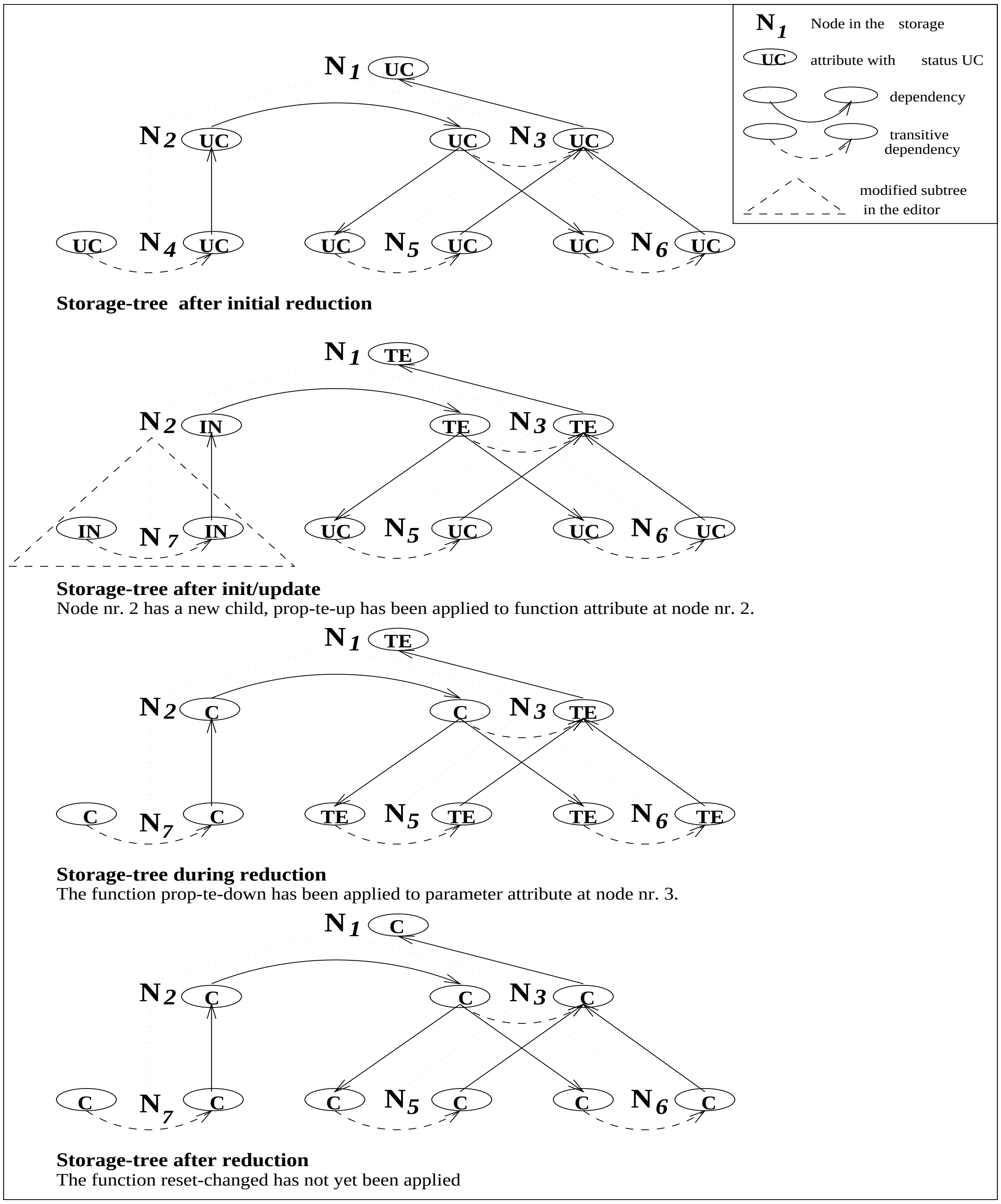


Figure 7.1: The status of the attributes in the storage-tree at various stages.

Example 7.5 The third storage-tree in Figure 7.1 show the effect of applying the function prop-te-down to the parameter attribute at node N_3. □

When a new value of an attribute has been evaluated, and this new value is equal to the value stored in the attribute, there is a chance that new values for its successors do not need to be evaluated. If the status of a successor is "TobeEvaluated" and it has only "UnChanged" predecessors, its status can be set to UC (UnChanged). When a new value is stored and this situation occur, the function funprop-unchanged is applied.

Attribute updating after reduction

When the reduction process has ended, the function reset-changed should be applied to reset the status of all "Changed" attributes to "UnChanged", and the status of their direct "TobeEvaluated" successors as well as their corresponding attributes to "Initial". For efficiency reasons, we add all attributes that during the evaluation get the status C (Changed) to a list in the storage.

7.3 Initialising and updating the storage

In this section we will discuss the way the storage is initialised and updated during editing. The storage is initialised and updated by means of a function init that is applied to the term in the editor when changes have been made in this term. The function init is derived from the original standard specification and a module InitUpdate is created as described in Chapter 6. Below we show the result of deriving init equations corresponding to the equation [Tc7] in the Pico-typecheck module discussed in Section 2.3.2.

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) = \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad \text{[Tc7]}$$

$$\mathsf{init}(\mathit{SERIESvar},\ S_1) = \mathsf{init}(\mathit{SERIESvar},\ 1,\ S_1) \qquad \text{[IR4]}$$

$$\frac{\mathsf{false} : S_2 = \mathsf{update\text{-}needed}(1;\ 2,\ R,\ S_1)}{\mathsf{init}(\mathit{Stat};\ \mathit{Series},\ R,\ S_1) = S_2} \qquad \text{[In5a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\ 2,\ R,\ S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ S_2),\\ \quad S_4 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\ S_3),\\ \quad S_5 = \mathsf{init}(\mathit{Stat},\ 1,\ S_4),\\ \quad S_6 = \mathsf{init}(\mathit{Series},\ 2,\ S_5)\end{array}}{\begin{array}{l}\mathsf{init}(\mathit{Stat};\ \mathit{Series},\ R,\ S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 1\\ \qquad \rightarrow \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 1.\\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \rightarrow \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2,\ S_6)\end{array}} \qquad \text{[In5b]}$$

The equation [IR4] is used to set the "rank" (the 1) when the initialisation or update begins. The "1" and the "2" in init(Stat,1,S_4) and init(Series,2,S_5), equation [In5b], are the ranks of these children. The numbers after each attribute in the dependencies also denotes ranks, "'0" being the "parent" node. [In5a] applies if no update is needed and [In5b] when a (sub) term in the abstract

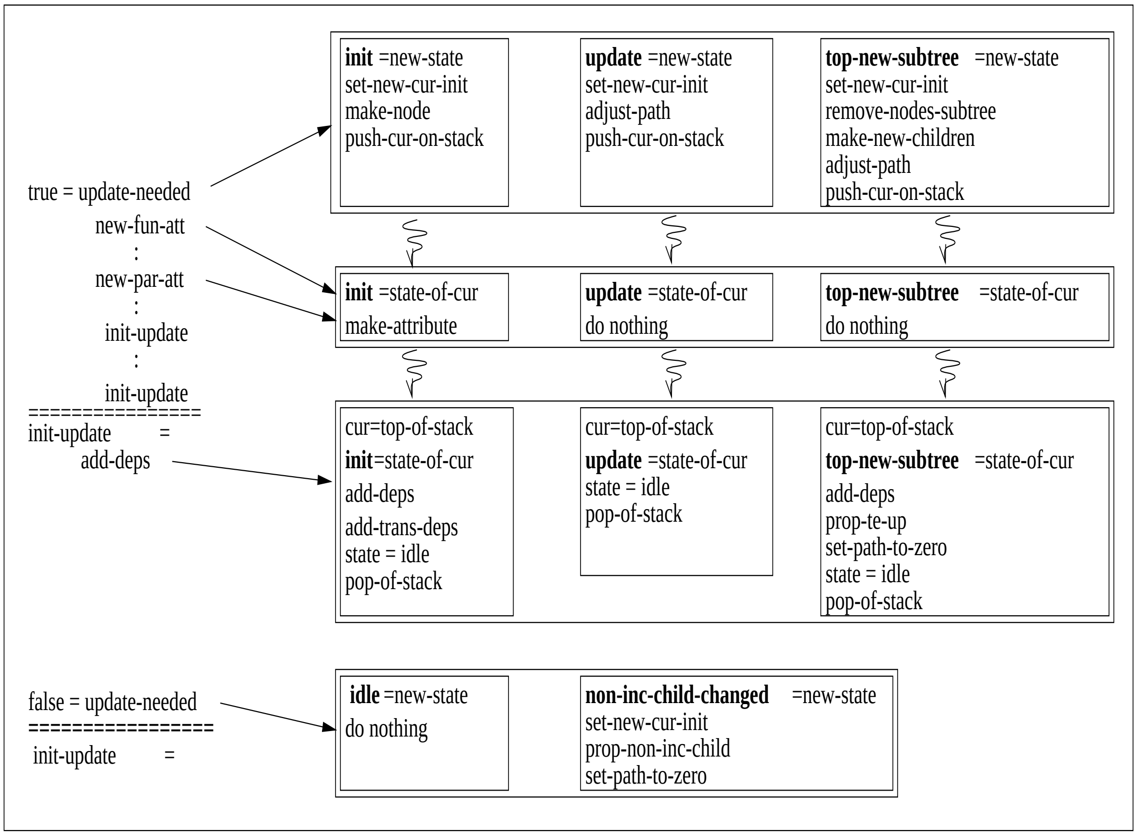


Figure 7.2: Overview of functions and side-effect depending on the state of the node.

syntax tree to which the storage corresponds has been modified. The four storage management functions used during initialisation and update are thus: update-needed, new-fun-att, new-par-att and add-deps. The global scheme of the function init is when update is needed:

- Add function and parameter attributes.
- Initialise or update the children.
- Add the dependencies to the attributes.

The functions update-needed and add-deps have side effects thus the global scheme just given is not the whole truth. In this section we will discuss the storage management functions. In Figure 7.2 we give an overview of the effects of the functions applied in equations such as [In5a] and [In5b] that is given above.

Before the storage can be initialised or updated knowledge of which part of the storage that need to be changed is required. In Section 6.6 we described how a module can be derived to find a path to the node in the storage that must be changed. A path is a list of ranks that can be followed from the top of the storage to find the part that must be replaced. The path must be provided to the storage before the update process begins. When the path has been supplied to the storage it is possible to determine the state of the *next current node*. In Section 7.3.1 we describe how this is done followed by a description of the function set-new-cur-init in Section 7.3.2. In Section 7.3.4 we will describe the actions that are taken depending on the state of a node.

7.3.1 Determine state "next" current node

When we introduced the nodes we use in the storage in Section 5.2.1 we only mentioned that a node can have various states. The state of the node indicates what "update" actions should be taken. The following states may occur: init, update, top-new-subtree, idle and non-inc-child-changed. The first condition in the equations that are derived to initialise and update the storage invokes the function update-needed. In the equations defining this function the first condition is used to determine what state the "next" current node should have by means of the function new-state. The previous current node is not needed to determine the state of the "next" current node. The function new-state needs the ranks of the children of the "next" current node, the rank of that node and the path. Below the equations defining the function new-state are given.

$$\text{new-state}(\mathit{NodeR}, , \mathit{ChildrenRs}) = \text{init} \qquad \text{[NS1]}$$

$$\frac{\mathit{NodeR} = \mathit{PathR}_1, \quad \text{true} = \text{present}(\mathit{PathR}_1, \mathit{ChildrenRs})}{\text{new-state}(\mathit{NodeR}, \mathit{PathR}_1; \mathit{PathR}_2; \mathit{PathRs}, \mathit{ChildrenRs}) = \text{update}} \qquad \text{[NS2]}$$

$$\frac{\mathit{NodeR} = \mathit{PathR}_1}{\text{new-state}(\mathit{NodeR}, \mathit{PathR}_1, \mathit{ChildrenRs}) = \text{top-new-subtree}} \qquad \text{[NS3]}$$

$$\frac{\mathit{NodeR} \neq \mathit{PathR}_1}{\text{new-state}(\mathit{NodeR}, \mathit{PathR}_1; \mathit{PathRs}, \mathit{ChildrenRs}) = \text{idle}} \qquad \text{[NS4]}$$

$$\frac{\mathit{NodeR} = \mathit{PathR}_1, \quad \text{false} = \text{present}(\mathit{PathR}_2, \mathit{ChildrenRs})}{\text{new-state}(\mathit{NodeR}, \mathit{PathR}_1; \mathit{PathR}_2; \mathit{PathRs}, \mathit{ChildrenRs}) = \text{non-inc-child-changed}} \qquad \text{[NS5]}$$

If the path is empty, equation [NS1], the storage must be initialised. When the rank of the "next" current node equals the first rank of the path and the second rank in the path is present in the list of children ranks, one of these children must be updated, [NS2]. In [NS3] the rank of the "next" current is equal to the path (the path consists of one rank), this node corresponds to the top of the new subtree in the abstract syntax tree. In equation [NS4] the rank of the node is not equal to the first rank in the path meaning no updating or initialisation is needed. As described in Section 5.3 no nodes are present in the storage corresponding to nodes in the abstract syntax tree for which no incremental functions have been declared. In [NS5] the first rank in the path is equal to the rank of the node but the second rank in the path is not present in the list with the ranks of the children, thus a non-incremental child has been changed.

7.3.2 Set new current node during init/update

In this section we will describe how the new current node is set in the storage during initialisation and update. This is done by invoking the function set-new-cur-init with the rank of the "next" current node. We can distinguish three cases: the stack in the storage is not empty, the stack is empty and the path is empty as well, or the stack is empty and the path is not. If the stack is not empty, the child with rank R of the node referred to on top of the stack is set to be the new current node. If the stack is empty the initialisation or update has just been started. If the path is empty, meaning that the whole storage must be re-initialised, a new storage is created. When the path is not empty the top of the storage is chosen as the new current node. The equations defining the function set-new-cur-init are given below.

$$\frac{\begin{array}{r}\text{level-of}(\text{SD}) = 0, \\ \text{path-of}(\text{SD}) = , \\ \text{SD}_1 : \mathit{Ns}_1 : \mathit{As}_1 = \text{make-storage}(\,), \\ \text{SD}_2 = \text{set-cur}(\text{top-of}(\text{SD}_1), \text{SD}_1)\end{array}}{\text{set-new-cur-init}(R, \text{SD} : \mathit{Ns} : \mathit{As}) = \text{SD}_2 : \mathit{Ns}_1 : \mathit{As}_1} \qquad \text{[SNCI1]}$$

$$\frac{\begin{array}{r}\textsf{level-of(SD)} = 0,\\ \textsf{path-of(SD)} \neq ,\\ \textsf{set-cur(top-of(SD), SD)} = \textsf{SD}_1\end{array}}{\textsf{set-new-cur-init}(R,\ \textsf{SD} : Ns : As) = \textsf{SD}_1 : Ns : As} \qquad [\textsf{SNCI2}]$$

$$\frac{\begin{array}{rl}\textsf{level-of(SD)} \neq 0, & \\ \textsf{NP}_1 = & \textsf{top-of-stack(SD)},\\ \textsf{NP}_2 = & \textsf{select-child}(R,\ \textsf{NP}_1,\ Ns),\\ \textsf{SD}_1 = & \textsf{set-cur}(\textsf{NP}_2,\ \textsf{SD})\end{array}}{\textsf{set-new-cur-init}(R,\ \textsf{SD} : Ns : As) = \textsf{SD}_1 : Ns : As} \qquad [\textsf{SNCI3}]$$

The variables in these equations: *SD* are the storage directions, *Ns* are nodes, *As* are attributes, *R* is the rank and *NP* are node pointers (the unique node numbers). The function level-of returns the number of objects on the stack.

7.3.3 Example

In this section we will give an example of an updating of the storage. In Figure 7.3 this process is shown. In 1, the update has not started yet, the stack is empty and the path is 1;2. The state of all nodes in the storage is idle. In 2. the current node is node nr 1. This node has rank one,

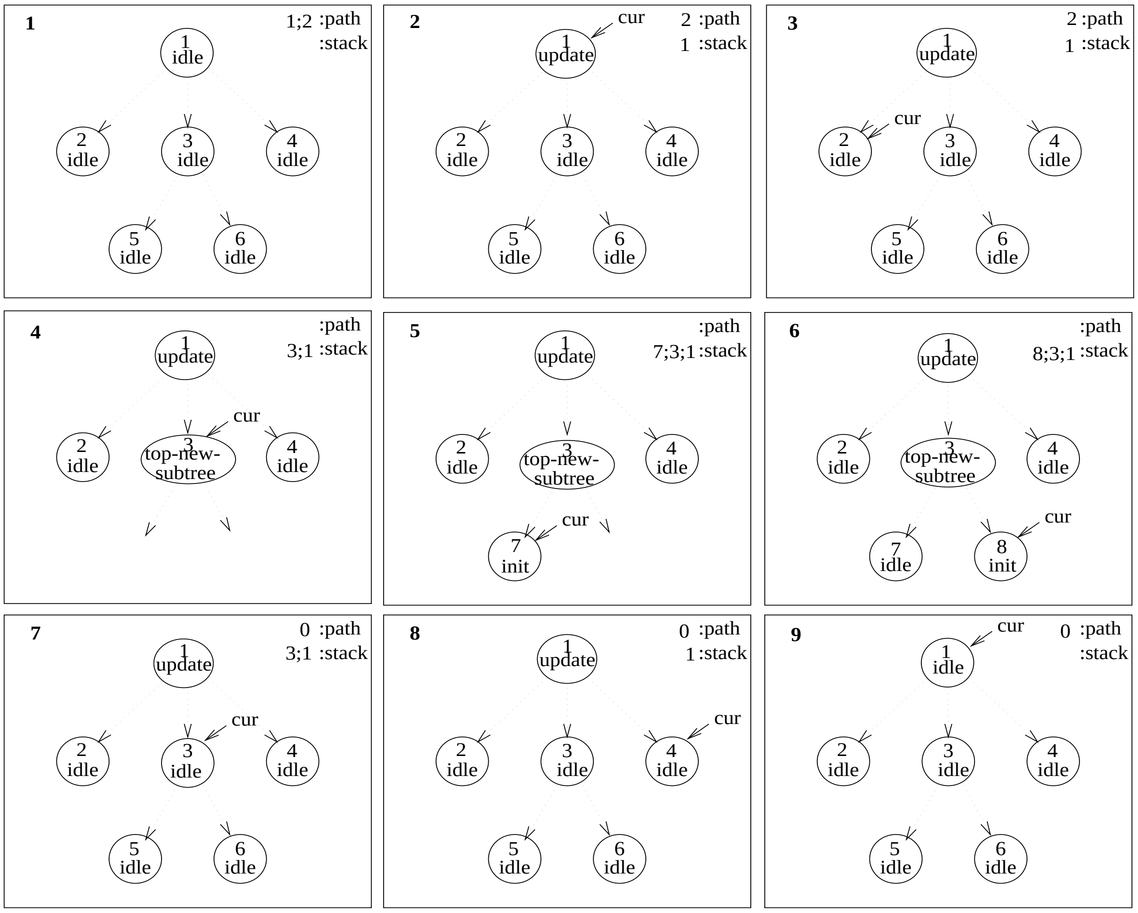


Figure 7.3: The status of the attributes in the storage at various stages during update.

the first rank in the path. Its state is set to update and a reference to this node is pushed on the stack. In 3, the current node is node number 2. Its rank (1) is not equal to the first rank in the path. Its state remains idle. In 4, the current node is node number 3. It has rank 2, the same rank as the first and only rank in the path. Its status is set to top-new-subtree and its children are removed and a reference to this node is pushed on the stack. In 5 and 6 the new children of node 3 are initialised and in 7 the current node is node number 3. Its status is set to idle, the path is set to 0 and the reference to this node is popped of the stack. In 8, the current node is node number 4. The path is 0, thus its state remains idle. In the last picture the state of node number 1 has been set to idle and the reference to this node has been pushed of the stack. The initialisation and update is completed.

7.3.4 Actions depending on the state of the node

In this section we will describe the actions depending on the state of the current node. The state of a node is used to determine what actions should be taken when the function add-deps has been invoked after the children of a node have been updated or initialised; when we return to a node. The new current node is found on top of the stack when add-deps is invoked; the last action in the functions update-needed(=true) was to push the current node on the stack. This is the way a reference to the current node is maintained. When the functions new-fun-att and new-par-att are invoked the current node has not been changed; there will be no additional conditions between the update-needed conditions and these function invocations in the equations defining the function init. The function update-needed has three arguments: the ranks of the children of the "next" current node *Rs*, the rank of the "next" current node *R* and the storage: *SD* (storage directions), *Ns* (nodes) and *As* (attributes).

State: init

When the function update-needed is invoked and the function new-state yields init the new current node is set by means of function set-new-cur-init. A new node is created by means of the function make-node. When a node is created its state is set to init. The max in the storage is increased with the number of children for which references were made when the node was created, the number of ranks in the variable *Rs*. Thus the references to nodes are created before the actual nodes are created. The current node is pushed on the stack by means of function push-cur-on-stack. The equation [UN1] defining the function update-needed when new-state=init is given below.

$$\frac{\begin{array}{r@{\;}l} \mathsf{init} &= \mathsf{new\text{-}state}(R,\ \mathsf{path\text{-}of}(\mathsf{SD}),\ Rs), \\ \mathsf{SD}_1 : Ns_1 : As_1 &= \mathsf{set\text{-}new\text{-}cur\text{-}init}(R,\ \mathsf{SD} : Ns : As), \\ N &= \mathsf{make\text{-}node}(\mathsf{cur\text{-}of}(\mathsf{SD}_1),\ \mathsf{init},\ \mathsf{max\text{-}of}(\mathsf{SD}_1),\ Rs), \\ \mathsf{SD}_2 &= \mathsf{set\text{-}new\text{-}max}(Rs,\ \mathsf{SD}_1), \\ \mathsf{SD}_3 &= \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_2) \end{array}}{\mathsf{update\text{-}needed}(Rs,\ R,\ \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_3 : Ns_1;\ N : As_1} \quad [\mathsf{UN1}]$$

When the functions new-fun-att and new-par-att are invoked and the state of the current node is init a new attribute is created by means of function make-att. Below the equation defining the function new-fun-att is given. The equation defining the function new-par-att for the same state of the current node looks the same but in the third condition fun (function attribute) is replaced by par (parameter attribute). Variables in the equation: *NP* is a node pointer, a unique node number, *NC* is the node content, *ANs* is a list of attribute names.

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} : \mathsf{NC};\ Ns_1 &= \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}),\ Ns), \\ \mathsf{init} &= \mathsf{state\text{-}of}(\mathsf{NP} : \mathsf{NC}), \\ A &= \mathsf{make\text{-}att}(\mathsf{fun},\ \mathsf{AN},\ \mathsf{NP}) \end{array}}{\mathsf{new\text{-}fun\text{-}att}(\mathsf{AN},\ \mathsf{SD} : Ns : As) = \mathsf{SD} : Ns : As;\ A} \quad [\mathsf{NFA}]$$

When the function add-deps is applied, the new current node is always on top of the stack in the storage. When the state of this node is init dependencies between attributes attached to the current node and to the current node's children are added by means of the function add-new-deps.

Transitive dependencies between the node's parameter attributes and function attributes are added by means of the function add-trans-deps. The definitions of these functions can be found in module Dependencies in Appendix H.4. When the dependencies have been added the state of the node is set to idle and the stack in the storage is popped. Superfluous attributes (see Section 6.4.5) that might have been added to the current nodes children are removed. The equation [AD1] defining the function add-deps is given below.

$$\frac{\begin{aligned} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack(SD)}, \\ N;\, Ns_1 &= \mathsf{find}(\mathsf{NP},\, Ns), \\ \mathsf{init} &= \mathsf{state\text{-}of}(N), \\ As_1 &= \mathsf{add\text{-}new\text{-}deps}(N,\, Deps,\, As), \\ As_2 &= \mathsf{add\text{-}trans\text{-}deps}(N,\, As_1), \\ N_1 &= \mathsf{set\text{-}state}(\mathsf{idle},\, N), \\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack(SD)}, \\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP},\, \mathsf{SD}_1) \end{aligned}}{\begin{aligned} &\mathsf{add\text{-}deps}(Deps,\, \mathsf{SD} : Ns : As) = \\ &\mathsf{rm\text{-}superfluous\text{-}atts}(\mathsf{SD}_2 : Ns_1;\, N_1 : As_2) \end{aligned}} \quad \text{[AD1]}$$

State: update

When the new state of a node is update one of the nodes children needs to be updated. The state of the (just found) current node is set to update. The first rank in the path is removed by means of the function remove-first-path-rank and the current node is pushed on the stack. Below the equation [UN2] is given, defining these actions.

$$\frac{\begin{aligned} \mathsf{update} &= \mathsf{new\text{-}state}(R,\, \mathsf{path\text{-}of(SD)},\, Rs), \\ \mathsf{SD}_1 : Ns_1 : As_1 &= \mathsf{set\text{-}new\text{-}cur\text{-}init}(R,\, \mathsf{SD} : Ns : As), \\ N;\, Ns_2 &= \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_1),\, Ns_1), \\ N_1 &= \mathsf{set\text{-}state}(\mathsf{update},\, N), \\ \mathsf{SD}_2 &= \mathsf{remove\text{-}first\text{-}path\text{-}rank}(\mathsf{SD}_1), \\ \mathsf{SD}_3 &= \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_2) \end{aligned}}{\mathsf{update\text{-}needed}(Rs,\, R,\, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_3 : N_1;\, Ns_2 : As} \quad \text{[UN2]}$$

When the functions new-fun-att and new-par-att are invoked and the state of the current node is not equal to init nothing is done. The equations are both default equations and are not presented.

The equation [AD2] given below applies when the state of the current node (found on the top of the stack) is update. No dependencies have to be added. Only the stack is popped.

$$\frac{\begin{aligned} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack(SD)}, \\ N;\, Ns_1 &= \mathsf{find}(\mathsf{NP},\, Ns), \\ \mathsf{update} &= \mathsf{state\text{-}of}(N), \\ N_1 &= \mathsf{set\text{-}state}(\mathsf{idle},\, N), \\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack(SD)}, \\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP},\, \mathsf{SD}_1) \end{aligned}}{\mathsf{add\text{-}deps}(Deps,\, \mathsf{SD} : Ns : As) = \mathsf{SD}_2 : Ns_1;\, N_1 : As} \quad \text{[AD2]}$$

State: top-new-subtree

If the state of the new current node is top-new-subtree all the subtrees of the current node are removed by means of the function remove-nodes-subtree. When this function is applied, dependencies between attributes in the current node and the attributes belonging to the children of this node are also removed. New children are added to the node by means of function new-children and the max in the storage is adjusted. The state of the current node is set to top-new-subtree and the current node is pushed on the stack. The path is adjusted: from now on, all the subtrees of the current node must be initialised, thus the path is empty. The equation [UN3] given below defines these actions.

$$\frac{\begin{array}{r@{\;=\;}l}\mathsf{top\text{-}new\text{-}subtree} & \mathsf{new\text{-}state}(R,\ \mathsf{path\text{-}of}(\mathsf{SD}),\ Rs),\\ \mathsf{SD}_1 : Ns_1 : As_1 & \mathsf{set\text{-}new\text{-}cur\text{-}init}(R,\ \mathsf{SD} : Ns : As),\\ \mathsf{SD}_2 : Ns_2 : As_2 & \mathsf{remove\text{-}nodes\text{-}subtree}(\mathsf{SD}_1 : Ns_1 : As_1),\\ N;\ Ns_3 & \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_2),\ Ns_2),\\ N_1 & \mathsf{new\text{-}children}(Rs,\ \mathsf{max\text{-}of}(\mathsf{SD}_2),\ N),\\ \mathsf{SD}_3 & \mathsf{set\text{-}new\text{-}max}(Rs,\ \mathsf{SD}_2),\\ N_2 & \mathsf{set\text{-}state}(\mathsf{top\text{-}new\text{-}subtree},\ N_1),\\ \mathsf{SD}_4 & \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_3),\\ \mathsf{SD}_5 & \mathsf{set\text{-}path}(,\ \mathsf{SD}_4)\end{array}}{\mathsf{update\text{-}needed}(Rs,\ R,\ \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_5 : Ns_3;\ N_2 : As_2} \quad \text{[UN3]}$$

No function or parameter attributes are added when the state of the current node is top-new-subtree; the node has not been changed, only its children.

Below the equation [AD3] is given defining the function add-deps when the state of the node on top of the stack is top-new-subtree. The status of all function attributes attached to the current node are set to IN (INitial)[2]. The function prop-te-up is applied to all these function attributes and the dependencies are added by means of the function add-new-deps. The transitive dependencies have not been removed (dependencies between parameter attributes and their corresponding function attributes) and need not be added. The path is set to 0 indicating that nothing no other parts of the storage needs updating. This prevents that siblings of the current node are updated. We only allow one new subtree thus the function init must be applied every time a subtree has been replaced in the editor. New variables: a node N consists of to parts; A unique number (the node pointer) NP and the content of the node NC, NP:NC.

$$\frac{\begin{array}{r@{\;=\;}l}\mathsf{NP}' & \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ \mathsf{NP} : \mathsf{NC};\ Ns_1 & \mathsf{find}(\mathsf{NP}',\ Ns),\\ \mathsf{top\text{-}new\text{-}subtree} & \mathsf{state\text{-}of}(\mathsf{NP} : \mathsf{NC}),\\ APs & \mathsf{fun\text{-}atts}(\mathsf{NP},\ As),\\ As_1 & \mathsf{set\text{-}status}(\mathsf{IN},\ APs,\ As),\\ As_2 & \mathsf{prop\text{-}te\text{-}up}(APs,\ As_1),\\ As_3 & \mathsf{add\text{-}new\text{-}deps}(\mathsf{NP} : \mathsf{NC},\ Deps,\ As_2),\\ \mathsf{SD}_1 & \mathsf{set\text{-}path}(0,\ \mathsf{SD}),\\ N & \mathsf{set\text{-}state}(\mathsf{idle},\ \mathsf{NP} : \mathsf{NC}),\\ \mathsf{SD}_2 & \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}_1),\\ \mathsf{SD}_3 & \mathsf{set\text{-}cur}(\mathsf{NP},\ \mathsf{SD}_2)\end{array}}{\begin{array}{l}\mathsf{add\text{-}deps}(Deps,\ \mathsf{SD} : Ns : As) =\\ \mathsf{rm\text{-}superfluous\text{-}atts}(\mathsf{SD}_3 : Ns_1;\ N : As_3)\end{array}} \quad \text{[AD3]}$$

State: idle

When the state of the new current node is idle nothing is done.

$$\frac{\mathsf{idle} = \mathsf{new\text{-}state}(R,\ \mathsf{path\text{-}of}(\mathsf{SD}),\ Rs)}{\mathsf{update\text{-}needed}(Rs,\ R,\ \mathsf{SD} : Ns : As) = \mathsf{false} : \mathsf{SD} : Ns : As} \quad \text{[UN4]}$$

State: non-inc-child-changed

This is a special case. In Section 5.3 we described what we mean by non-incremental children. The problem is that other (incremental) children can be dependent on the result of the reduction of this non-incremental child. In [Meu94] this problem was solved by introducing "*fake*" attributes, so called *syntactic references*. In Figure 7.4 this situation is illustrated. Because we are able to

[2]Their values are not consistent with the values of their predecessors.

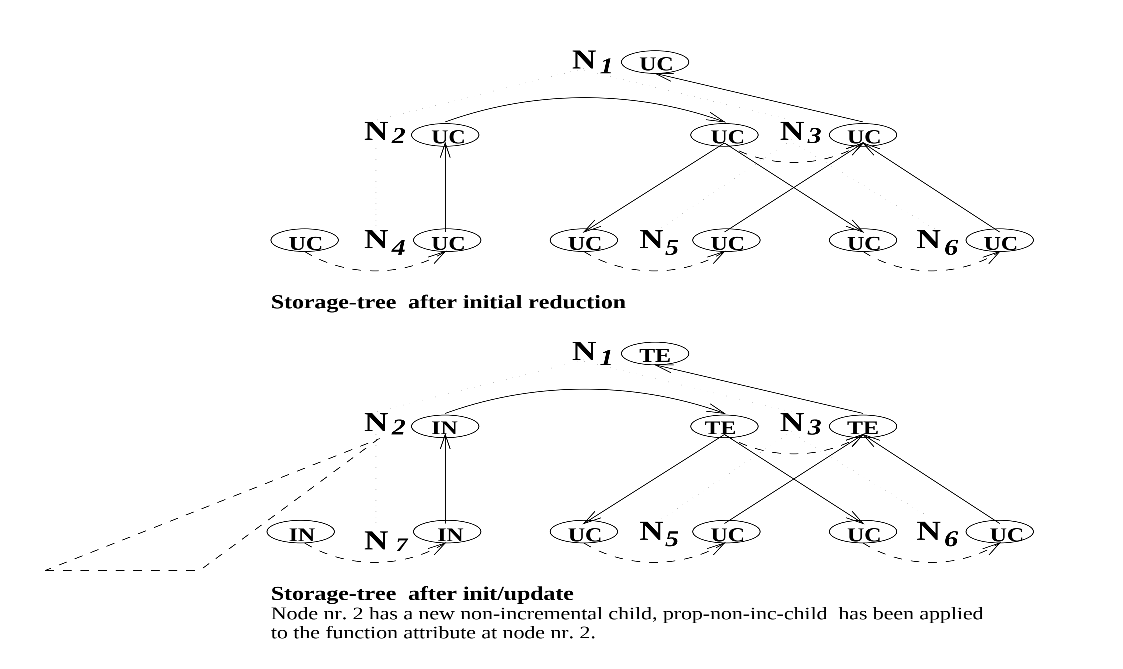


Figure 7.4: The status of the attributes in the storage at various stages when a non incremental child of N_2 has been changed.

detect these children, we only need to define a new attribute propagation function. We call this function prop-non-inc-child and the equation defining this function is given below.

$$\frac{\begin{array}{rl} \mathit{APs} = & \text{fun-atts}(\text{NP}, \mathit{As}), \\ \mathit{As}_1 = & \text{set-status}(\text{IN}, \mathit{APs}, \mathit{As}), \\ \mathit{NPs} = & \text{children-of}(\text{NP} : \text{NC}), \\ \mathit{As}_2 = & \text{set-status}(\text{IN}, \mathit{NPs}, \mathit{As}_1), \\ \mathit{As}_3 = & \text{prop-te-up}(\mathit{APs}, \mathit{As}_2) \end{array}}{\text{prop-non-inc-child}(\text{NP} : \text{NC}, \mathit{As}) = \mathit{As}_3} \qquad \text{[PNIC1]}$$

First attribute pointers (APs) to all the function attributes attached to the current node are selected by means of function fun-atts. The status of all these function attributes are set to IN by means of the function set-status and thereafter the function prop-te-up is applied to these function attributes. The node numbers of the children of the node (NP:NC, the combination of node number and node content representing a node) are selected and all attributes attached to these children are set to IN.

When the state of a new current node is non-inc-child-changed, the node refered to by the current node pointer is selected by means of function find and the function prop-non-inc-child that we just described is applied to reset the status of attributes which value might be dependent on this term. In equation [UN5] given below these actions are defined.

$$\frac{\begin{array}{rl} \text{non-inc-child-changed} = & \text{new-state}(R, \text{path-of}(\text{SD}), \mathit{Rs}), \\ \text{SD}_1 : \mathit{Ns}_1 : \mathit{As}_1 = & \text{set-new-cur-init}(R, \text{SD} : \mathit{Ns} : \mathit{As}), \\ N;\ \mathit{Ns}_2 = & \text{find}(\text{cur-of}(\text{SD}_1), \mathit{Ns}_1), \\ \mathit{As}_2 = & \text{prop-non-inc-child}(N, \mathit{As}_1), \\ \text{SD}_2 = & \text{set-path}(0, \text{SD}_1) \end{array}}{\text{update-needed}(\mathit{Rs}, R, \text{SD} : \mathit{Ns} : \mathit{As}) = \text{false} : \text{SD}_2 : \mathit{Ns}_1 : \mathit{As}_2} \qquad \text{[UN5]}$$

7.4 Storage management during reduction

In this section we will discuss the storage management functions that are used during incremental reduction with incremental specifications. When module(s) are transformed into incremental modules, conditions are added to the conditions of the original equations. These conditions invoke storage management functions. Below we show the result of transforming the equation [Tc7] in the Pico-typecheck module discussed in Section 2.3.2.

$$\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv}) \;=\; \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv})\ \&\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) \qquad \text{[Tc7]}$$

$$\mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ S) \;=\; \mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ 1:0,\ S) \qquad \text{[AII4]}$$

$$\frac{\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}safe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R:L,\ S)}{\begin{array}{l}\mathsf{tcs}(\mathit{SERIESvar},\ \mathit{Tenv},\ R:L,\ S) =\\ \mathsf{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ S_1)\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS4]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R:L,\ S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\ |\ \mathit{Tenv}\ |\ ,\ S_1),\\ |\ \mathit{BOOLvar}_1\ | : S_3 = \mathsf{tcstat}(\mathit{Stat},\ \mathit{Tenv},\ 1:L+1,\ S_2),\\ |\ \mathit{BOOLvar}_2\ | : S_4 = \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv},\ 2:L+1,\ S_3)\end{array}}{\begin{array}{l}\mathsf{tcs}(\mathit{Stat};\ \mathit{Series},\ \mathit{Tenv},\ R:L,\ S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\\ \qquad |\ \mathit{BOOLvar}_1\ \&\ \mathit{BOOLvar}_2\ |\ ,\ S_3)\end{array}} \qquad \text{[ITc7]}$$

Equation [AII4] is applied when the reduction starts. In this equation our INCINFO is initialised (1:0). It consists of two parts: a rank R and a level L (R:L). In Section 5.4 we explained why this additional information is needed. Roughly speaking, the rank R informs us which child of the current node should be the next to be visited and the level L keeps track of how many objects there should be on the stack in the storage. Equation [SIS4] states that if the status of the function attributed `"tcs(SERIES,TENV)->BOOL"` is safe the value stored in this attribute can be retrieved and reused. Equation [ITc7] applies when the status of this attribute is unsafe. The value of the parameter *Tenv* is converted into a normal form by putting it between bars and this normal form is stored in the parameter attribute `"par(tcs(SERIES,TENV)->BOOL,1)"`. The functions tcstat and tcs are applied. The rank of these subterms was set during the transformation: *Stat* and *Series* respectively represents the first and the second child of *Stat;Series*. This rank is combined with the level L that is increased by one; a term at one *level deeper* will be reduced. Finally the normal form of the function is stored in the function attribute by means of the function store-fun-att. The storage management functions used during reduction are: status-is-safe, status-is-unsafe, store-par-att, store-fun-att and retrieve.

The storage management function invoked during reduction are not as complex as the ones discussed in the prior section. We start by giving a description of the function set-new-cur that sets the new current followed by the functions first invoked in the conditions in an incremental specification: status-is-safe and status-is-unsafe. We will then continue with the function retrieve that retrieves a normal form in the storage. We end this section with the functions that stores new values: store-par-att and store-fun-att.

7.4.1 Set the current node during reduction

We will now define the function set-new-cur. Equation [SNC1] is applied if the level L in the term that is being reduced is zero; the reduction has just started. We can not be sure that the level (number of objects on the stack) in the storage has the same value for reason given in Section 5.4. The level in the storage is set to 0 and the top of the storage is set as new current node.

$$\frac{\begin{array}{r}\mathsf{set\text{-}level}(0, \mathsf{SD}) = \mathsf{SD}_1,\\ \mathsf{set\text{-}cur}(\mathsf{top\text{-}of}(\mathsf{SD}_1), \mathsf{SD}_1) = \mathsf{SD}_2,\\ \mathsf{set\text{-}stack}(, \mathsf{SD}_2) = \mathsf{SD}_3\end{array}}{\mathsf{set\text{-}new\text{-}cur}(R : 0, \mathsf{SD}, Ns) = \mathsf{SD}_3} \qquad \text{[SNC1]}$$

$$\frac{\begin{array}{r}L \neq 0,\\ \mathsf{level\text{-}of}(\mathsf{SD}) = L,\\ N; Ns_1 = \mathsf{find}(\mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}), Ns),\\ \mathsf{NP} = \mathsf{select\text{-}child\text{-}ptr}(R, N),\\ \mathsf{SD}_1 = \mathsf{set\text{-}cur}(\mathsf{NP}, \mathsf{SD})\end{array}}{\mathsf{set\text{-}new\text{-}cur}(R : L, \mathsf{SD}, Ns) = \mathsf{SD}_1} \qquad \text{[SNC2]}$$

$$\frac{\begin{array}{r}L \neq \mathsf{level\text{-}of}(\mathsf{SD}),\\ \mathsf{SD}_1 = \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD})\end{array}}{\begin{array}{l}\mathsf{set\text{-}new\text{-}cur}(R : L, \mathsf{SD}, Ns) =\\ \mathsf{set\text{-}new\text{-}cur}(R : L, \mathsf{SD}_1, Ns)\end{array}} \qquad \text{[SNC3]}$$

The equation [SNC2] and [SNC3] are applied if the level L in the term does not equal zero. If the level in the storage does not equal L, equation [SNC3] the stack is popped until they are equal and equation [SNC2] is applied. The top of the stack is now the previous current. The child with rank R of the previous current is selected by means of function select-child-ptr. This node is set as the new current.

7.4.2 Check the status

The function status-is-safe checks the status of an attribute. When the new current node has been set, the attribute with name AN is selected and the status of this attribute is checked. If the status is safe, true is returned. The equations defining the function status-is-safe is given below.

$$\frac{\begin{array}{r}\mathsf{SD}_1 = \mathsf{set\text{-}new\text{-}cur}(R : L, \mathsf{SD}, Ns),\\ A; As_1 = \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_1) : \mathsf{AN}, As),\\ \mathsf{true} = \mathsf{safe\text{-}att}(A)\end{array}}{\mathsf{status\text{-}is\text{-}safe}(\mathsf{AN}, R : L, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_1 : Ns : As} \qquad \text{[SiS]}$$

$$\frac{\begin{array}{r}\mathsf{SD}_1 = \mathsf{set\text{-}new\text{-}cur}(R : L, \mathsf{SD}, Ns),\\ A; As_1 = \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_1) : \mathsf{AN}, As),\\ \mathsf{false} = \mathsf{safe\text{-}att}(A),\\ \mathsf{SD}_2 = \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_1)\end{array}}{\mathsf{status\text{-}is\text{-}unsafe}(\mathsf{AN}, R : L, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_2 : Ns : As} \qquad \text{[SiU]}$$

The function status-is-unsafe is defined by equation [SiU] above. This function also checks the status of an attribute. When the status is unsafe the term must be reduced. This means that the current node must be located at a later stage during the reduction and is thus pushed on the stack and the value true is returned.

7.4.3 Retrieve value

When the function retrieve is invoked, the last storage management function invoked was status-is-safe. In status-is-safe the current node has been set and has not been changed since. The current node is used to locate the attribute with name AN and the normal form stored in this attribute is returned. A reference to an attribute consists of the number of the node to which the attribute is attached and an attribute name. The function cur-of yields such a number and this number combined with the attribute name AN is a unique combination and the attribute can be found.

$$\frac{\begin{array}{l} A;\, As_1 = \mathsf{find}(\mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}) : \mathsf{AN},\, As), \\ \quad Nf = \mathsf{nf\text{-}of}(A) \end{array}}{\mathsf{retrieve}(\mathsf{AN},\, \mathsf{SD} : Ns : As) = Nf : \mathsf{SD} : Ns : As} \qquad \text{[Ret1]}$$

7.4.4 Store parameter attributes

When the function store-par-att is invoked, the last storage management function invoked was the function status-is-unsafe by which the new current node has been set. The current node can thus be used to find the attribute with name AN. If the status of the attribute is safe nothing is done, the default. When the status of the attribute is unsafe the normal form that should be stored is compared with the normal form the attribute contains. If these normal forms are equal, the status of the attribute is set to UC (UnChanged) and the attribute propagation function prop-unchanged is applied to this attribute. If these two normal forms are not equal, equation [SPA2], the new normal form is stored in the attribute and its status is set to C (Changed). For reasons described in Section 7.2 the attribute propagations function prop-te-down is applied to the attribute. The attribute just got status C (Changed) and is also added to the list of changed attributes in the storage by means of the function add-changed-atts.

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} &= \mathsf{cur\text{-}of}(\mathsf{SD}), \\ A;\, As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\, As), \\ \mathsf{C} &\neq \mathsf{status\text{-}of}(A), \\ Nf &= \mathsf{nf\text{-}of}(A), \\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{UC},\, A), \\ As_2 &= \mathsf{prop\text{-}unchanged}(\mathsf{AP},\, As_1;\, \mathsf{AP} : \mathsf{AC}) \end{array}}{\mathsf{store\text{-}par\text{-}att}(\mathsf{AN},\, Nf,\, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD} : Ns : As_1;\, \mathsf{AP} : \mathsf{AC}} \qquad \text{[SPA1]}$$

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} &= \mathsf{cur\text{-}of}(\mathsf{SD}), \\ A;\, As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\, As), \\ \mathsf{C} &\neq \mathsf{status\text{-}of}(A), \\ Nf &\neq \mathsf{nf\text{-}of}(A), \\ A_1 &= \mathsf{set\text{-}nf}(Nf,\, A), \\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{C},\, A_1), \\ \mathsf{SD}_1 &= \mathsf{add\text{-}changed\text{-}att}(\mathsf{AP},\, \mathsf{SD}), \\ As_2 &= \mathsf{prop\text{-}te\text{-}down}(\mathsf{AP},\, As_1;\, \mathsf{AP} : \mathsf{AC}) \end{array}}{\mathsf{store\text{-}par\text{-}att}(\mathsf{AN},\, Nf,\, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_1 : Ns : As_1;\, \mathsf{AP} : \mathsf{AC}} \qquad \text{[SPA2]}$$

$$\mathsf{store\text{-}par\text{-}att}(\mathsf{AN},\, Nf,\, S) \;=\; \mathsf{true} : S \quad \textbf{otherwise} \qquad \text{[SPA]}$$

7.4.5 Store function attributes

When the function store-fun-att is applied, the current node is on the top of the stack in the storage. By using the top of the stack the attribute with the name AN can be found. If the normal form stored in the attribute is not equal to the normal form Nf that is supposed to be stored, equation [SFA1] the status of the attribute is set to C (Changed). A reference to this attribute is added to the list of changed attributes by means of the function add-changed-att. A subtree in the corresponding abstract syntax tree has now been evaluated which means that we can remove the reference to the current node on the stack by popping it. When the normal form Nf equals the normal form stored in the attribute, equation [SF2], the status of the attribute is set to UC (UnChanged) and the attribute propagation function prop-unchanged is applied to the attribute and for the same reasons as above, the stack can be popped.

$$\frac{\begin{array}{rl} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ A;\ As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\ As),\\ Nf &\neq \mathsf{nf\text{-}of}(A),\\ A_1 &= \mathsf{set\text{-}nf}(Nf,\ A),\\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{C},\ A_1),\\ \mathsf{SD}_1 &= \mathsf{add\text{-}changed\text{-}att}(\mathsf{AP},\ \mathsf{SD}),\\ \mathsf{SD}_2 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}_1),\\ \mathsf{SD}_3 &= \mathsf{set\text{-}cur}(\mathsf{NP},\ \mathsf{SD}_2) \end{array}}{\mathsf{store\text{-}fun\text{-}att}(\mathsf{AN},\ Nf,\ \mathsf{SD} : Ns : As) = Nf : \mathsf{SD}_3 : Ns : As_1;\ \mathsf{AP} : \mathsf{AC}} \quad \text{[SF1]}$$

$$\frac{\begin{array}{rl} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ A;\ As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\ As),\\ Nf &= \mathsf{nf\text{-}of}(A),\\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{UC},\ A),\\ As_2 &= \mathsf{prop\text{-}unchanged}(\mathsf{AP},\ As_1;\ \mathsf{AP} : \mathsf{AC}),\\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}),\\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP},\ \mathsf{SD}_1) \end{array}}{\mathsf{store\text{-}fun\text{-}att}(\mathsf{AN},\ Nf,\ \mathsf{SD} : Ns : As) = Nf : \mathsf{SD}_2 : Ns : As_2} \quad \text{[SF2]}$$

7.4.6 Incremental reduction: Example

Assume that the equation [Tc12] and [Tc13] are part of a specification that is transformed into an incremental specification. These two equations defines the same function tcexpr and let us say that this function has been declared as incremental.

$$\frac{Exp = Str\text{-}con}{\mathsf{tcexpr}(Exp,\ Tenv) = \mathsf{string}} \quad \text{[Tc12]}$$

$$\frac{Exp = Nat\text{-}con}{\mathsf{tcexpr}(Exp,\ Tenv) = \mathsf{natural}} \quad \text{[Tc13]}$$

When these equations are transformed the result will be the following four equations: [AII6] to initialise the rank and the level, [SiS6] is the safe equation and [ITc12] and [ITc13] will be applied when the status of the attribute is unsafe. In this example we use the equations that will be generated when the external functions can be used, thus the storage is not part of the term reduced.

$$\mathsf{tcexpr}(EXPvar,\ Tenv)\ =\ \mathsf{tcexpr}(EXPvar,\ Tenv,\ 1 : 0) \quad \text{[AII6]}$$

$$\frac{\mathsf{true} = \mathsf{status\text{-}is\text{-}safe}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ R : L)}{\begin{array}{l}\mathsf{tcexpr}(EXPvar,\ Tenv,\ R : L) =\\ \mathsf{nf2TYPE}(\mathsf{retrieve}(\texttt{"tcexpr(EXP,TENV)->TYPE"}))\end{array}} \quad \text{[SIS6]}$$

$$\frac{\begin{array}{l} \mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\ |\ Tenv\ |),\\ Exp = Nat\text{-}con,\\ \ \ Nf = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ |\ \mathsf{natural}\ |) \end{array}}{\mathsf{tcexpr}(Exp,\ Tenv,\ R : L) = \mathsf{nf2TYPE}(Nf)} \quad \text{[ITc12]}$$

$$\frac{\begin{array}{l}\text{true} = \text{status-is-unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \text{true} = \text{store-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, |\, \mathit{Tenv} \,|),\\ \mathit{Exp} = \mathit{Str\text{-}con},\\ \mathit{Nf} = \text{store-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, |\, \text{string} \,|)\end{array}}{\text{tcexpr}(\mathit{Exp}, \mathit{Tenv}, R : L) = \text{nf2TYPE}(\mathit{Nf})} \quad \text{[ITc13]}$$

Assume that the attributes `"tcexpr(EXP,TENV)->TYPE"` and `"par(tcexpr(EXP,TENV)->TYPE,1)"` have status TE (TobeEvaluated). When the reduction starts with a term, the rewrite engine tries to find a match with the left-hand side of an equation. Assume that equation [SIS6] is tried. The first condition will fail because the function attribute is TE and thus not safe. Let say the rewrite engine then tries equation [ITc12]. The first condition will succeed, the status of the attribute is unsafe. In the next condition, the parameter attribute is stored. Let us assume that the value stored in this parameter attribute equals the value that is stored. In this case a propagation on the attributes is done; prop-te-up and both the parameter attribute `"par(tcexpr(EXP,TENV)->TYPE,1)"` as well as the function attribute `"tcexpr(EXP,TENV)->TYPE"` gets status UC. If the third condition in equation [ITc12] fails equation [ITc13] will be tried. The status of the attribute has however been changed and is safe thus the condition fails. Equation [SIS6] will not be tried once more and the incremental reduction fails.

7.4.7 No defaults

To prevent that situations as described in the previous section occur the “safe” equations are default equations and will always be tried as the last one. This is thus the reason why we do not allow default equation defined for incremental functions in the modules that are to be transformed.

7.5 After reduction

In the module SM-after-reduction given below we have defined the last storage management function reset-changed-atts.

module SM-after-reduction
imports Propagations$^{H.1}$
exports
  context-free syntax
    reset-changed-atts(STORAGE) → STORAGE
equations

The function reset-changed-atts defined below should be called between reduction. This function selects all attributes in the storage that have status Changed, and changes their status to UnChanged by means of the function funreset-changed that is defined in the module Propagations.

$$\frac{\begin{array}{r}\text{ch-atts-of}(\text{SC}) = \mathit{APs},\\ \mathit{As}_1 = \text{reset-changed}(\mathit{APs}, \mathit{As}),\\ \text{SC}_1 = \text{reset-ch-atts}(\text{SC})\end{array}}{\text{reset-changed-atts}(\text{SC} : \mathit{Ns} : \mathit{As}) = \text{SC}_1 : \mathit{Ns} : \mathit{As}_1} \quad \text{[rca]}$$

Chapter 8

Conclusions

8.1 Summary

We have developed a technique for incremental rewriting based on transforming specifications into so-called incremental specifications. In these specifications conditions are introduced to invoke functions for storing and retrieving normal forms. The incremental specifications are independent of the implementation of the rewrite engine as long as the rewrite strategy is innermost. We have specified the transformations in ASF+SDF for ASF+SDF specifications in ASFIX notation to derive these incremental specification.

A storage structure has been specified in ASF+SDF for storing and retrieving normal forms during reduction. The storage has been designed in such a manner that time consuming search actions are avoided by supplying the storage with directions indicating which node in the storage should be the next current node. Storage management functions have been specified for manipulating the storage during initialisation or update of the storage, during reduction and after reduction. These functions are invoked in conditions in equations in the incremental specification and supply the storage with the direction information. These functions return values that have influence on the operation of the rewrite engine.

The definition of a Well-presented Primitive recursive schemes has been studied and a minumum of properties have been chosen that a specification must satisfy to enable incremental rewriting with incremental specifications. Properties that do not have any influence on the incremental rewriting but imposes restrictions on the way a user should write his specification have been relaxed. A check on the properties has been specified in ASF+SDF. The result of the PRS-check is a list of detailed error messages in ASFIX. The produced messages concerning the kind of error are rather brief. We have, however, specified functions that produce more detailed messages.

We have in ASF+SDF specified transformations of modules denoted in ASFIX to enable invocation of the storage management functions. There are three different transformations:

- For each module an incremental module is derived. This module should be used during the reduction.
- For all modules one InitUpdate module is derived. This module should be used before reduction (during editing) to initialise or update the storage.
- For all modules one PathSearch module is derived. This approach forces us to keep a copy of the term for which the storage was initialised the last time to determine where in the storage update or initialisation is needed.

8.2 Discussion

One of the most important obstacles we faced was the limitations of the hybrid functions in the current implementation of ASF+SDF. It would have been nice to be able to test the incremental rewriting at a level were the storage exists at the implementation level rather than as part of the term that is reduced. The efficiency of incremental rewriting with incremental specifications is strongly dependent on the external C-functions that will be supported shortly. Of course, the number of conditions that have to be evaluated during reduction with incremental specification has increased, and the conversion itself must also be taken into account. When a term is reduced, 3 steps will always be taken when the attribute corresponding to the top operator of that term is reliable. If this attribute is not reliable, the number of steps that should be taken during non-incremental reduction will be taken plus 4 extra steps for checking the status of the attribute and storing the new normal form. If the term has parameters, 2 extra steps are needed to store the normal form of each parameter in its corresponding parameter attribute.

8.3 Future work

The storage and the storage management functions should be implemented in a conventional programming language to establish an increase in efficiency with respect to non-incremental rewriting. An interface should be created between ASF+SDF functions and external functions. For this purpose a storage interface should be created, a link between the function invocations in the equations in the incremental specification and the storage management functions.

Example 8.1 The future definition of the function status-is-safe in a StorageInterface module when external C function are supported by the implementation of the system. The function after "EXTC" is a normal C-function.

$$\frac{\mathit{Bool} = \mathsf{EXTC\ status\_is\_safe(AN,IncInfo,get\_storage())}}{\mathsf{status\text{-}is\text{-}safe(AN,}\ \mathit{IncInfo}\mathsf{)} = \mathit{Bool}} \quad \text{[SIS]}$$

□

The user of the ASF+SDF Meta-environment might use more than one term editor in a session. Therefore there should be a link between a storage and its corresponding editor. In Figure 8.1 we give our view of the system when parts of the incremental specification have been integrated in the system. During editing it should be possible to let the system compute the path to the new subtree in the editor and set this path in the storage that corresponds to an editor in which a subtree replacement has been done. It should also be possible to let a compiled version of the InitUpdate module be integrated in the system to update or initialise a storage during editing. After reduction of a term, the system could also invoke the function to reset changed attributes in the storage.

The check whether a specification is a PRS can be compiled into C code by means of the ASF2C-compiler [KW93]. This is also possible for the transformation of modules. It should be possible to combine these compiled modules in a Unix script, in which modules can be transformed to ASFIX then checked and when no errors are found, transformed into incremental modules.

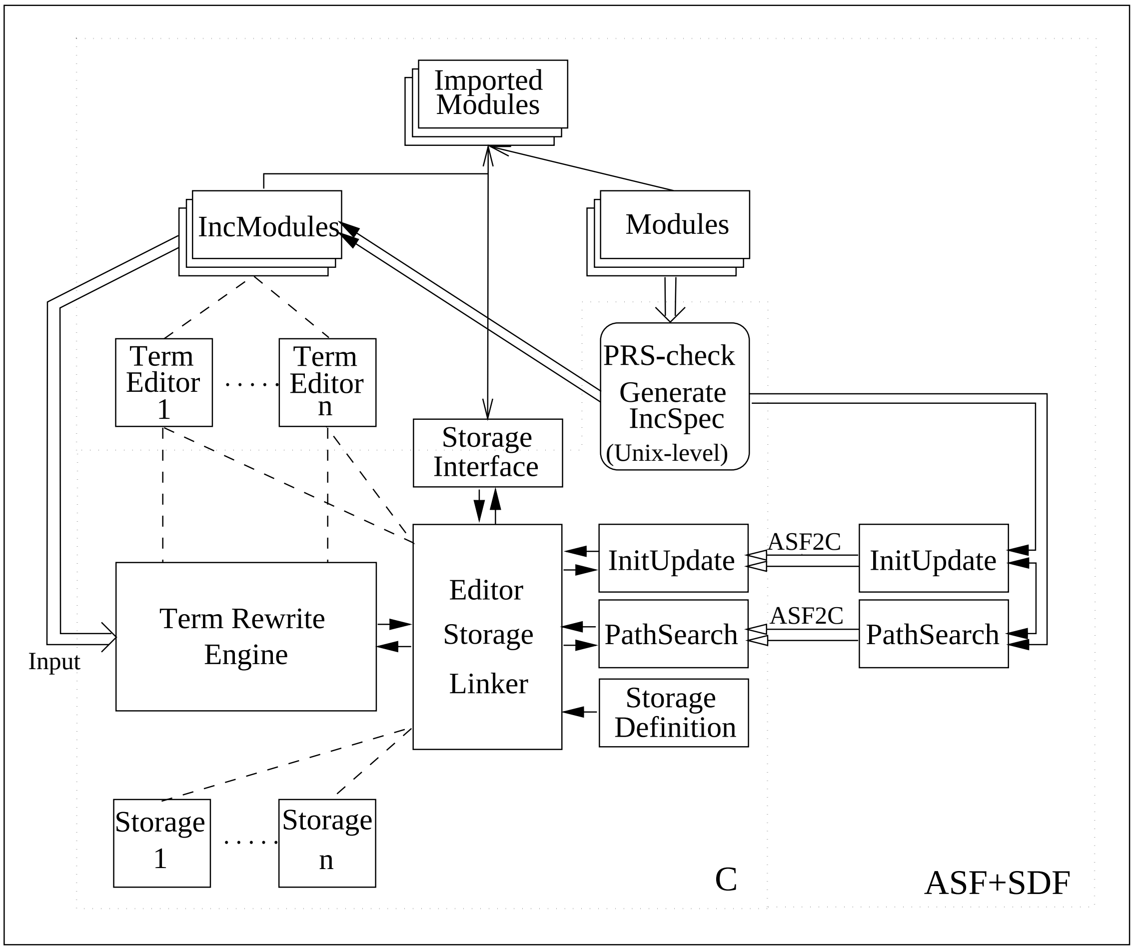


Figure 8.1: Our view of the integration of incremental specifications in a rewrite system.

Bibliography

[BHK89a] J.A. Bergstra, J. Heering and P. Klint. *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.

[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.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers. *The Syntax Definition Formalism SDF, reference manual* 1992.

[Kli93b] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176-201, 1993.

[KW93] J.F.T. Kamperman and H.R. Walters. *ARM, abstract rewrite machine.* Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993.

[Kli94] P. Klint. Writing meta-level specifications in ASF+SDF. 1994.

[Meu94] E.A. van der Meulen. *Incremental Rewriting.* PhD thesis, University of Amsterdam, 1994.

Appendix A

Auxiliary data-structures

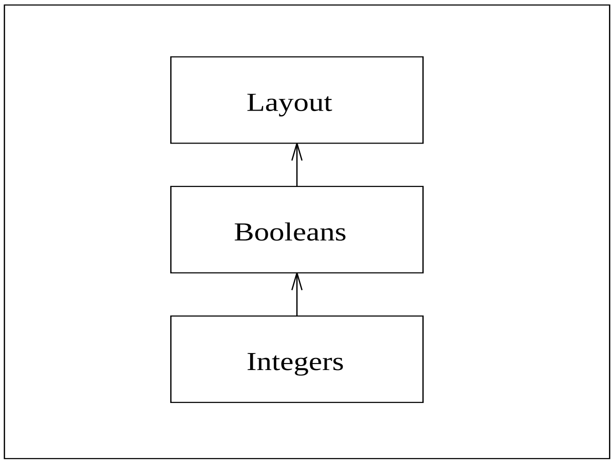


Figure A.1: Module import overview.

A.1 Layout

module Layout
exports
  lexical syntax
```
    "%%" ~[\n]*[\n] → LAYOUT
    "%" ~[\n]* "%"  → LAYOUT
    [␣\t\n]         → LAYOUT
```

A.2 Booleans

module Booleans
imports Layout$^{A.1}$
exports
  sorts BOOL
  context-free syntax
```
    true   → BOOL
    false  → BOOL
```

BOOL "|" BOOL → BOOL {**left**}
BOOL "&" BOOL → BOOL {**left**}
not "(" BOOL ")" → BOOL
"(" BOOL ")" → BOOL {**bracket**}
variables
Bool [*0-9′*]∗ → BOOL
priorities
"|" < "&"

equations

true | *Bool*	=	true	[B1]
false | *Bool*	=	*Bool*	[B2]
true & *Bool*	=	*Bool*	[B3]
false & *Bool*	=	false	[B4]
not(false)	=	true	[B5]
not(true)	=	false	[B6]

A.3 Integers

module Integers
imports Booleans[A.2]
exports
sorts NAT-CON NAT INT
lexical syntax
[0-9]+ → NAT-CON
context-free syntax
NAT-CON → NAT
NAT "−/" NAT → NAT {**left**}
"(" NAT ")" → NAT {**bracket**}
NAT → INT
"+" NAT → INT
"−" NAT → INT
INT "+" INT → INT {**left**}
INT "−" INT → INT {**left**}
INT "∗" INT → INT {**left**}
INT ">" INT → BOOL
INT "≥" INT → BOOL
INT "<" INT → BOOL
INT "≤" INT → BOOL
"(" INT ")" → INT {**bracket**}
variables
Nat-con [*0-9′*]∗ → NAT-CON
Nat [*0-9′*]∗ → NAT
Int [*0-9′*]∗ → INT
hiddens
context-free syntax
gt "(" NAT "," NAT ")" → BOOL

NAT "$-//$" NAT $\rightarrow$ NAT
NAT "$\succ$" NAT $\rightarrow$ NAT

variables

n[0-9]$*$ $\rightarrow$ NAT-CON
c[0-9]$*$ $\rightarrow$ CHAR
x[0-9]$*$ $\rightarrow$ CHAR$*$
y[0-9]$*$ $\rightarrow$ CHAR+

priorities

{**left**: INT "+"INT $\rightarrow$ INT, INT "$-$"INT $\rightarrow$ INT} $<$ INT "$*$"INT $\rightarrow$ INT

equations

[1] nat-con("0" *y*) $=$ nat-con(*y*)

– addition –

[2] $0 + \mathit{Nat} = \mathit{Nat}$

[3] $\mathit{Nat} + 0 = \mathit{Nat}$

[4]	$1 + 1 = 2$	[5]	$1 + 2 = 3$	[6]	$1 + 3 = 4$
[7]	$1 + 4 = 5$	[8]	$1 + 5 = 6$	[9]	$1 + 6 = 7$
[10]	$1 + 7 = 8$	[11]	$1 + 8 = 9$	[12]	$1 + 9 = 10$
[13]	$2 + 1 = 3$	[14]	$2 + 2 = 4$	[15]	$2 + 3 = 5$
[16]	$2 + 4 = 6$	[17]	$2 + 5 = 7$	[18]	$2 + 6 = 8$
[19]	$2 + 7 = 9$	[20]	$2 + 8 = 10$	[21]	$2 + 9 = 11$
[22]	$3 + 1 = 4$	[23]	$3 + 2 = 5$	[24]	$3 + 3 = 6$
[25]	$3 + 4 = 7$	[26]	$3 + 5 = 8$	[27]	$3 + 6 = 9$
[28]	$3 + 7 = 10$	[29]	$3 + 8 = 11$	[30]	$3 + 9 = 12$
[31]	$4 + 1 = 5$	[32]	$4 + 2 = 6$	[33]	$4 + 3 = 7$
[34]	$4 + 4 = 8$	[35]	$4 + 5 = 9$	[36]	$4 + 6 = 10$
[37]	$4 + 7 = 11$	[38]	$4 + 8 = 12$	[39]	$4 + 9 = 13$
[40]	$5 + 1 = 6$	[41]	$5 + 2 = 7$	[42]	$5 + 3 = 8$
[43]	$5 + 4 = 9$	[44]	$5 + 5 = 10$	[45]	$5 + 6 = 11$
[46]	$5 + 7 = 12$	[47]	$5 + 8 = 13$	[48]	$5 + 9 = 14$
[49]	$6 + 1 = 7$	[50]	$6 + 2 = 8$	[51]	$6 + 3 = 9$
[52]	$6 + 4 = 10$	[53]	$6 + 5 = 11$	[54]	$6 + 6 = 12$
[55]	$6 + 7 = 13$	[56]	$6 + 8 = 14$	[57]	$6 + 9 = 15$
[58]	$7 + 1 = 8$	[59]	$7 + 2 = 9$	[60]	$7 + 3 = 10$
[61]	$7 + 4 = 11$	[62]	$7 + 5 = 12$	[63]	$7 + 6 = 13$
[64]	$7 + 7 = 14$	[65]	$7 + 8 = 15$	[66]	$7 + 9 = 16$
[67]	$8 + 1 = 9$	[68]	$8 + 2 = 10$	[69]	$8 + 3 = 11$
[70]	$8 + 4 = 12$	[71]	$8 + 5 = 13$	[72]	$8 + 6 = 14$
[73]	$8 + 7 = 15$	[74]	$8 + 8 = 16$	[75]	$8 + 9 = 17$
[76]	$9 + 1 = 10$	[77]	$9 + 2 = 11$	[78]	$9 + 3 = 12$
[79]	$9 + 4 = 13$	[80]	$9 + 5 = 14$	[81]	$9 + 6 = 15$
[82]	$9 + 7 = 16$	[83]	$9 + 8 = 17$	[84]	$9 + 9 = 18$

[85a] $$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c), \quad \text{nat-con}(y_1) + \text{nat-con}(y_2) + \text{nat-con}(\text{"0"}\ x) = \text{nat-con}(y)}{\text{nat-con}(y_1\ \ c_1) + \text{nat-con}(y_2\ \ c_2) = \text{nat-con}(y\ c)}$$

[85b] $$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c), \quad \text{nat-con}(\text{"0"}\ x_1) + \text{nat-con}(y_2) + \text{nat-con}(\text{"0"}\ x) = \text{nat-con}(y)}{\text{nat-con}(x_1\ \ c_1) + \text{nat-con}(y_2\ \ c_2) = \text{nat-con}(y\ c)}$$

[85c] $$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c), \quad \text{nat-con}(y_1) + \text{nat-con}(\text{"0"}\ x_2) + \text{nat-con}(\text{"0"}\ x) = \text{nat-con}(y)}{\text{nat-con}(y_1\ \ c_1) + \text{nat-con}(x_2\ \ c_2) = \text{nat-con}(y\ c)}$$

– auxiliary function -// for cut off subtraction

[86] $\text{nat-con}(c) -// \text{nat-con}(c) = 0$

[87] $\text{nat-con}(c) -// 0 = \text{nat-con}(c)$

[88] $2 -// 1 = 1$

[89] $3 -// 1 = 2$ [90] $3 -// 2 = 1$

[91] $4 -// 1 = 3$ [92] $4 -// 2 = 2$ [93] $4 -// 3 = 1$

[94] $5 -// 1 = 4$ [95] $5 -// 2 = 3$ [96] $5 -// 3 = 2$

[97] $5 -// 4 = 1$

[98] $6 -// 1 = 5$ [99] $6 -// 2 = 4$ [100] $6 -// 3 = 3$

[101] $6 -// 4 = 2$ [102] $6 -// 5 = 1$

[103] $7 -// 1 = 6$ [104] $7 -// 2 = 5$ [105] $7 -// 3 = 4$

[106] $7 -// 4 = 3$ [107] $7 -// 5 = 2$ [108] $7 -// 6 = 1$

[109] $8 -// 1 = 7$ [110] $8 -// 2 = 6$ [111] $8 -// 3 = 5$

[112] $8 -// 4 = 4$ [113] $8 -// 5 = 3$ [114] $8 -// 6 = 2$

[115] $8 -// 7 = 1$

[116] $9 -// 1 = 8$ [117] $9 -// 2 = 7$ [118] $9 -// 3 = 6$

[119] $9 -// 4 = 5$ [120] $9 -// 5 = 4$ [121] $9 -// 6 = 3$

[122] $9 -// 7 = 2$ [123] $9 -// 8 = 1$

[124] $10 -// 1 = 9$ [125] $10 -// 2 = 8$ [126] $10 -// 3 = 7$

[127] $10 -// 4 = 6$ [128] $10 -// 5 = 5$ [129] $10 -// 6 = 4$

[130] $10 -// 7 = 3$ [131] $10 -// 8 = 2$ [132] $10 -// 9 = 1$

– auxiliary function gt

[133] $$\frac{\text{nat-con}(c_1) -// \text{nat-con}(c_2) = \text{nat-con}(c), \quad \text{nat-con}(c) \neq 0}{\text{gt}(\text{nat-con}(c_1), \text{nat-con}(c_2)) = \text{true}}$$

[134] $\text{gt}(\text{nat-con}(y_1\ \ c_1), \text{nat-con}(c_2)) = \text{true}$

[135] $\text{gt}(\text{nat-con}(y\ c_1), \text{nat-con}(y\ c_2)) = \text{gt}(\text{nat-con}(c_1), \text{nat-con}(c_2))$

[136] $$\frac{\text{gt}(\text{nat-con}(y_1), \text{nat-con}(y_2)) = \text{true}}{\text{gt}(\text{nat-con}(y_1\ \ c_1), \text{nat-con}(y_2\ \ c_2)) = \text{true}}$$

– auxiliary function ¿- , partial substraction

[137] $Nat \succ 0 \;=\; Nat$

[138] $\text{nat-con}(c_1) \succ \text{nat-con}(c_2) \;=\; \text{nat-con}(c_1) -// \text{nat-con}(c_2)$

[139a] $$\frac{\text{nat-con}(c_1) -// \text{nat-con}(c_2) = \text{nat-con}(c_3),\quad \text{nat-con}(y_1) \succ \text{nat-con}(\texttt{"0"}\ x_2) = \text{nat-con}(y)}{\text{nat-con}(y_1\ c_1) \succ \text{nat-con}(x_2\ c_2) = \text{nat-con}(y\ c_3)}$$

[139b] $$\frac{\text{nat-con}(c_2) -// \text{nat-con}(c_1) = \text{nat-con}(c_3),\quad 10 -// \text{nat-con}(c_3) = \text{nat-con}(c),\quad \text{nat-con}(\texttt{"0"}\ x_2) + 1 = Nat,\quad \text{nat-con}(y_1) \succ Nat = \text{nat-con}(y)}{\text{nat-con}(y_1\ c_1) \succ \text{nat-con}(x_2\ c_2) = \text{nat-con}(y\ c)}$$

– cut off subtraction -/

[140] $\text{gt}(Nat_1, Nat_2) = \text{true} \Rightarrow Nat_1 -/ Nat_2 \;=\; Nat_1 \succ Nat_2$

[141] $\text{gt}(Nat_1, Nat_2) \neq \text{true} \Rightarrow Nat_1 -/ Nat_2 \;=\; 0$

– subtraction of naturals

[142] $\text{gt}(Nat_1, Nat_2) = \text{true} \Rightarrow Nat_1 - Nat_2 \;=\; Nat_1 \succ Nat_2$

[143] $\text{gt}(Nat_1, Nat_2) \neq \text{true} \Rightarrow Nat_1 - Nat_2 \;=\; - Nat_2 \succ Nat_1$

– multiplication of naturals –

[144] $Nat * 0 \;=\; 0$

[145] $Nat * 1 \;=\; Nat$

[146] $$\frac{\text{gt}(\text{nat-con}(c), 1) = \text{true}}{Nat * \text{nat-con}(c) = Nat + Nat * (\text{nat-con}(c) - 1)}$$

[147] $$\frac{\text{nat-con}(y_1) * \text{nat-con}(y_2) = \text{nat-con}(y)}{\text{nat-con}(y_1) * \text{nat-con}(y_2\ c) = \text{nat-con}(y\ \texttt{"0"}) + \text{nat-con}(y_1) * \text{nat-con}(c)}$$

– addition, subtraction, and multiplication of integers

[148] $+ Nat \;=\; Nat$

[149] $Nat_1 + - Nat_2 \;=\; Nat_1 - Nat_2$

[150] $- Nat_1 + Nat_2 \;=\; Nat_2 - Nat_1$

[151] $Nat_1 + Nat_2 = Nat \Rightarrow - Nat_1 + - Nat_2 \;=\; - Nat$

[152] $Nat_1 - - Nat_2 \;=\; Nat_1 + Nat_2$

[153] $Nat_1 + Nat_2 = Nat \Rightarrow - Nat_1 - Nat_2 \;=\; - Nat$

[154] $- Nat_1 - - Nat_2 \;=\; Nat_2 - Nat_1$

[155] $\mathit{Nat}_1 * \mathit{Nat}_2 = \mathit{Nat} \Rightarrow \mathit{Nat}_1 * -\ \mathit{Nat}_2 \ = \ -\ \mathit{Nat}$

[156] $\mathit{Nat}_1 * \mathit{Nat}_2 = \mathit{Nat} \Rightarrow -\ \mathit{Nat}_1 * \mathit{Nat}_2 \ = \ -\ \mathit{Nat}$

[157] $-\ \mathit{Nat}_1 * -\ \mathit{Nat}_2 \ = \ \mathit{Nat}_1 * \mathit{Nat}_2$

– relational operators

[158] $\mathsf{gt}(\mathit{Nat}_1, \mathit{Nat}_2) = \mathsf{true} \Rightarrow \mathit{Nat}_1 > \mathit{Nat}_2 \ = \ \mathsf{true}$

[159] $\mathsf{gt}(\mathit{Nat}_1, \mathit{Nat}_2) \neq \mathsf{true} \Rightarrow \mathit{Nat}_1 > \mathit{Nat}_2 \ = \ \mathsf{false}$

[160] $\mathit{Nat}_1 > -\ \mathit{Nat}_2 \ = \ \mathsf{true}$

[161] $-\ \mathit{Nat}_1 > \mathit{Nat}_2 \ = \ \mathsf{false}$

[162] $-\ \mathit{Nat}_1 > -\ \mathit{Nat}_2 \ = \ \mathit{Nat}_2 > \mathit{Nat}_1$

[163] $\mathit{Int}_1 \neq \mathit{Int}_2 \Rightarrow \mathit{Int}_1 \geq \mathit{Int}_2 \ = \ \mathit{Int}_1 > \mathit{Int}_2$

[164] $\mathit{Int} \geq \mathit{Int} \ = \ \mathsf{true}$

[165] $\mathit{Int}_1 < \mathit{Int}_2 \ = \ \mathsf{not}(\mathit{Int}_1 \geq \mathit{Int}_2)$

[166] $\mathit{Int}_1 \leq \mathit{Int}_2 \ = \ \mathsf{not}(\mathit{Int}_1 > \mathit{Int}_2)$

– no minus zero

[167] $-\ 0 \ = \ 0$

Appendix B

Pico

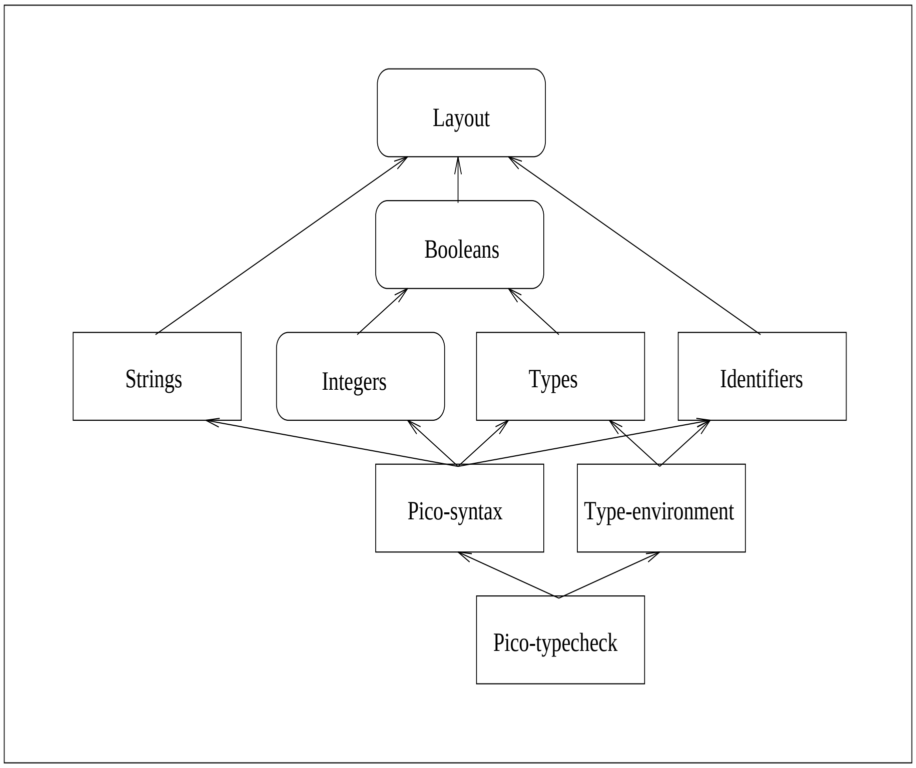


Figure B.1: Module import overview.

B.1 Identifiers

module Identifiers
imports Layout$^{A.1}$
exports
  sorts ID
  lexical syntax
    [a-z][a-z0-9]$*$ $\rightarrow$ ID
  variables
    Id [*0-9′*]$*$ $\rightarrow$ ID

B.2 Strings

module Strings
imports Layout$^{A.1}$
exports
sorts STR-CON STRING
lexical syntax
"\"" ~[\"\n]* "\"" → STR-CON
context-free syntax
STR-CON → STRING
STRING "||" STRING → STRING {**left**}
variables
Str-con [0-9']* → STR-CON
Str [0-9']* → STRING
hiddens
variables
Chars [12] → CHAR*
equations

str-con(""" $Chars_1$ """) || str-con(""" $Chars_2$ """) =
str-con(""" $Chars_1$ $Chars_2$ """) [S1]

B.3 Types

module Types
imports Booleans$^{A.2}$
exports
sorts TYPE
context-free syntax
natural → TYPE
string → TYPE
compatible "(" TYPE "," TYPE ")" → BOOL
variables
Type [0-9']* → TYPE
equations

compatible(natural, natural) = true [Typ1]
compatible(string, string) = true [Typ2]

B.4 Type-environments

module Type-environments
imports Identifiers$^{B.1}$ Types$^{B.3}$

exports
sorts TENV PAIR
context-free syntax
“(” 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 $[0\text{-}9']*$ $\rightarrow$ TENV
Pair $[0\text{-}9]*$ $\rightarrow$ PAIR
equations

$$\frac{\text{identifier of } \mathit{Pair} = \mathit{Id}}{\text{lookup } \mathit{Pair} + \mathit{Tenv} \text{ for } \mathit{Id} = \text{type of } \mathit{Pair}} \qquad [\text{T1}]$$

$$\frac{\text{identifier of } \mathit{Pair} \neq \mathit{Id}}{\begin{array}{l}\text{lookup } \mathit{Pair} + \mathit{Tenv} \text{ for } \mathit{Id} = \\ \text{lookup } \mathit{Tenv} \text{ for } \mathit{Id}\end{array}} \qquad [\text{T2}]$$

$$\text{identifier of } (\mathit{Id} : \mathit{Type}) \;=\; \mathit{Id} \qquad [\text{T3}]$$

$$\text{type of } (\mathit{Id} : \mathit{Type}) \;=\; \mathit{Type} \qquad [\text{T4}]$$

B.5 Pico-syntax

module Pico-syntax

imports Identifiers$^{B.1}$ Integers$^{A.3}$ Types$^{B.3}$ Strings$^{B.2}$
exports
sorts PROGRAM DECLS ID-TYPE-LIST ID-TYPE SERIES STATEMENT EXP
context-free syntax
begin DECLS SERIES end $\rightarrow$ PROGRAM
declare ID-TYPE-LIST “;” $\rightarrow$ DECLS
ID-TYPE “,” ID-TYPE-LIST $\rightarrow$ ID-TYPE-LIST
ID-TYPE $\rightarrow$ ID-TYPE-LIST
ID “:” TYPE $\rightarrow$ ID-TYPE

STATEMENT “;” SERIES $\rightarrow$ SERIES
STATEMENT $\rightarrow$ SERIES
ID “:=” EXP $\rightarrow$ STATEMENT
if EXP then SERIES else SERIES fi $\rightarrow$ STATEMENT
while EXP do SERIES od $\rightarrow$ STATEMENT

ID $\rightarrow$ EXP
NAT-CON $\rightarrow$ 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 [*0-9*]∗	→ ID-TYPE-LIST
Id-type [*0-9*]∗	→ ID-TYPE
Series [*0-9′*]∗	→ SERIES
Stat [*0-9′*]∗	→ STATEMENT
Exp [*0-9′*]∗	→ EXP

B.6 Pico-typecheck

¡¡hello.tex¿¿=

module Pico-typecheck

imports Booleans$^{A.2}$ Pico-syntax$^{B.5}$ Type-environments$^{B.4}$

exports

context-free syntax

tcidlist(ID-TYPE-LIST, TENV)	→ TENV	{**incremental**}
tcid(ID-TYPE)	→ PAIR	
tcexpr(EXP, TENV)	→ TYPE	{**incremental**}
tcp(PROGRAM)	→ BOOL	{**incremental**}
tcd(DECLS)	→ TENV	{**incremental**}
tcs(SERIES, TENV)	→ BOOL	{**incremental**}
tcstat(STATEMENT, TENV)	→ BOOL	{**incremental**}

@

equations

tcp(begin *Decls Series* end) = tcs(*Series*, tcd(*Decls*)) [Tc1]

tcd(declare *Id-type-list*;) = tcidlist(*Id-type-list*, empty-tenv) [Tc2]

tcidlist(*Id-type*, *Tenv*) = tcid(*Id-type*) + *Tenv* [Tc3]

tcidlist(*Id-type*, *Id-type-list*, *Tenv*) = tcidlist(*Id-type-list*, tcid(*Id-type*) + *Tenv*) [Tc4]

tcid(*Id* : *Type*) = (*Id* : *Type*) [Tc5]

tcs(*Stat*, *Tenv*) = tcstat(*Stat*, *Tenv*) [Tc6]

tcs(*Stat*; *Series*, *Tenv*) = tcstat(*Stat*, *Tenv*) & tcs(*Series*, *Tenv*) [Tc7]

tcstat(*Id* := *Exp*, *Tenv*) = compatible(lookup *Tenv* for *Id*, tcexpr(*Exp*, *Tenv*)) [Tc8]

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcs}(\mathit{Series}_1,\ \mathit{Tenv}) = \mathit{Bool}_1,\\ \mathsf{tcs}(\mathit{Series}_2,\ \mathit{Tenv}) = \mathit{Bool}_2\end{array}}{\mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\ \mathit{Tenv}) = \mathit{Bool}_1\ \&\ \mathit{Bool}_2} \quad [\mathsf{Tc9}]$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp},\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcs}(\mathit{Series},\ \mathit{Tenv}) = \mathit{Bool}\end{array}}{\mathsf{tcstat}(\mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od},\ \mathit{Tenv}) = \mathit{Bool}} \quad [\mathsf{Tc10}]$$

$$\mathsf{tcexpr}(\mathit{Id},\ \mathit{Tenv}) = \mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id} \quad [\mathsf{Tc11}]$$

$$\mathsf{tcexpr}(\mathit{Nat\text{-}con},\ \mathit{Tenv}) = \mathsf{natural} \quad [\mathsf{Tc12}]$$

$$\mathsf{tcexpr}(\mathit{Str\text{-}con},\ \mathit{Tenv}) = \mathsf{string} \quad [\mathsf{Tc13}]$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}} \quad [\mathsf{Tc14}]$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{natural},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 - \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{natural}} \quad [\mathsf{Tc15}]$$

$$\frac{\begin{array}{c}\mathsf{tcexpr}(\mathit{Exp}_1,\ \mathit{Tenv}) = \mathsf{string},\\ \mathsf{tcexpr}(\mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{string}\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 \parallel \mathit{Exp}_2,\ \mathit{Tenv}) = \mathsf{string}} \quad [\mathsf{Tc16}]$$

Appendix C

AsFix

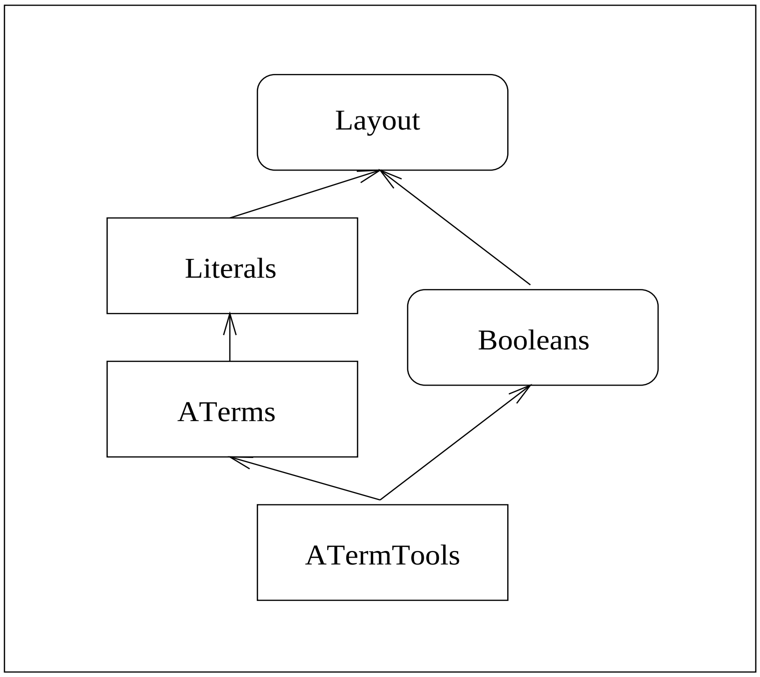


Figure C.1: Module import overview.

C.1 Literals

Literals are quoted strings constants. They serve the purpose of "constants" in the definition of ATerms.

module Literals
imports Layout[A.1]
exports
  sorts Literal
  lexical syntax
    "\\" ∼□ → EscChar
    "\\" [01][0-7][0-7] → EscChar

    ∼[\000-\037"\\] → L-Char

EscChar → L-Char
"\"" L-Char∗ "\"" → Literal

C.2 ATerms

An ATerm is either a Literal, a list construction (possibly the empty list nil), an application of an annotation.

module ATerms
imports Literals$^{C.1}$
exports
sorts ATerm
context-free syntax

Literal	→ ATerm	
ATerm ";" ATerm	→ ATerm	{**right**}
nil	→ ATerm	
"[" ATerm ATerm "]"	→ ATerm	
ATerm "/" ATerm	→ ATerm	{**left**}
"(" ATerm ")"	→ ATerm	{**bracket**}

variables

T [*0-9'*]∗	→ ATerm
Ts [*0-9'*]∗	→ {ATerm ","}∗
Ts "+" [*0-9'*]∗	→ {ATerm ","}+
L [*0-9'*]∗	→ Literal

priorities
ATerm "/" ATerm → ATerm > ATerm ";" ATerm → ATerm

C.3 ATermTools

In the module ATermTools two functions on ATerms are defined: occurs and merge. The function occurs is a Boolean function that determines whether a given term occurs as subterm in another term. The function merge merges two lists of ATerms and at the same time removes all appearances of the empty list nil.

module ATermTools
imports ATerms$^{C.2}$ Booleans$^{A.2}$
exports
context-free syntax
occurs(ATerm, ATerm) → BOOL
merge(ATerm, ATerm) → ATerm
equations
The function *occurs* below determines whether a given term appears as subterm in another term.

$$T \neq T_1;\, T_2 \Rightarrow \ \mathsf{occurs}(T,\, T_1;\, T_2) \ = \ \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad \text{[Occ1]}$$

$$T \neq [T_1\ T_2] \Rightarrow \ \mathsf{occurs}(T,\, [T_1\ T_2]) \ = \ \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad \text{[Occ2]}$$

$$T \neq T_1 \,/\, T_2 \Rightarrow \ \mathsf{occurs}(T,\, T_1 \,/\, T_2) \ = \ \mathsf{occurs}(T,\, T_1) \mid \mathsf{occurs}(T,\, T_2) \qquad \text{[Occ3]}$$

$$\mathsf{occurs}(T,\, T) \ = \ \mathsf{true} \qquad \text{[Occ4]}$$

$$\mathsf{occurs}(T,\, \mathsf{nil}) \ = \ \mathsf{false} \qquad \text{[Occ5]}$$

$$\mathsf{occurs}(T_1,\, T_2) \ = \ \mathsf{false} \quad \textbf{otherwise} \qquad \text{[Occ]}$$

The function *merge* below takes two list of ATerms as input and produces one list of ATerms. The function *merge* also removes empty (sub) lists (nil) out of the lists. The output produced by the function is a list of ATerms without any nil or the produced list contains only one ATerm, the empty list nil.

$$\mathsf{merge}(T,\ \mathsf{nil}) = T \qquad \text{[Merg1]}$$

$$\mathsf{merge}(\mathsf{nil},\ T) = T \qquad \text{[Merg2]}$$

$$\mathsf{merge}(\mathsf{nil};\ T_1,\ T_2) = \mathsf{merge}(T_1,\ T_2) \qquad \text{[Merg3]}$$

$$\mathsf{merge}(T;\ T_1,\ T_2) = T;\ \mathsf{merge}(T_1,\ T_2) \quad \textbf{when}\ \ T_1 \neq \mathsf{nil},\ \ T \neq \mathsf{nil} \qquad \text{[Merg4]}$$

$$\mathsf{merge}(T,\ T_1) = T;\ T_1 \quad \textbf{otherwise} \qquad \text{[Merg]}$$

Appendix D

Auxiliary AsFix

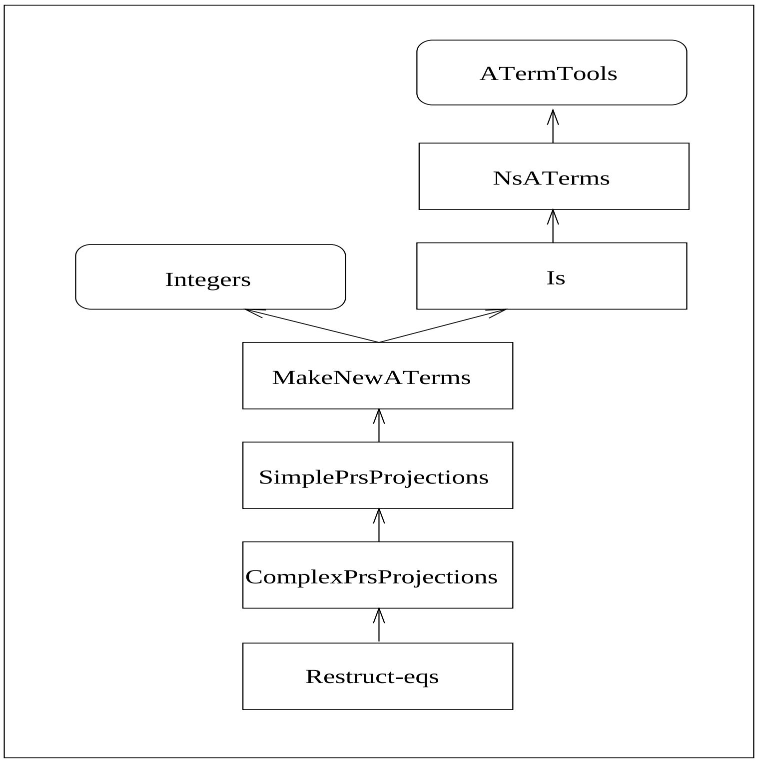


Figure D.1: Module import overview.

D.1 NsATerms

In the module NsATerms we have defined two functions: ns-aterms and aterms. The first function substitutes all predefined string constants by introduced constants of type Literal. This saves memory at reduction time and keeps the size down of the ATerms representing an ASF+SDF-module. If an ASF+SDF specification is compiled to C, the code will be more efficient if there are not to many string constants present in the specification. The function aterms transforms an (Ns)ATerm back into the normal AsFix representation.

module NsATerms
imports ATermTools$^{C.3}$

exports

context-free syntax

ns-aterms(ATerm)	→ ATerm
aterms(ATerm)	→ ATerm
scModule	→ Literal
scSourceSyntax	→ Literal
scPrefixSyntax	→ Literal
scImports	→ Literal
scExports	→ Literal
scHiddens	→ Literal
scSorts	→ Literal
scLexicalSyntax	→ Literal
scContextFreeSyntax	→ Literal
scPriorities	→ Literal
scVariables	→ Literal
scId	→ Literal
scSort	→ Literal
scLexicalFunction	→ Literal
scLexIter	→ Literal
scCharClass	→ Literal
scNeg	→ Literal
scLexElems	→ Literal
scQLit	→ Literal
scCfFunction	→ Literal
scCfElems	→ Literal
scIter	→ Literal
scAttributes	→ Literal
scIncrChain	→ Literal
scDecrChain	→ Literal
scAbbrev	→ Literal
scAbbrevs	→ Literal
scFunctionList	→ Literal
scVariable	→ Literal
scSourceEquations	→ Literal
scPrefixEquations	→ Literal
scWhen	→ Literal
scImplies	→ Literal
scCondEquation	→ Literal
scCondition	→ Literal
scNeq	→ Literal
scEq	→ Literal
scConditions	→ Literal
scVar	→ Literal
scLex	→ Literal
scincremental	→ Literal

equations

ns-aterms(T_1; T_2)	=	ns-aterms(T_1); ns-aterms(T_2)	[NsA1]
ns-aterms([T_1 T_2])	=	[ns-aterms(T_1) ns-aterms(T_2)]	[NsA2]
ns-aterms(T_1 / T_2)	=	ns-aterms(T_1) / ns-aterms(T_2)	[NsA3]
ns-aterms(`"Module"`)	=	scModule	[NsA4]

ns-aterms("SourceSyntax")	=	scSourceSyntax		[NsA5]
ns-aterms("PrefixSyntax")	=	scPrefixSyntax		[NsA6]
ns-aterms("Imports")	=	scImports		[NsA7]
ns-aterms("Exports")	=	scExports		[NsA8]
ns-aterms("Hiddens")	=	scHiddens		[NsA9]
ns-aterms("Sorts")	=	scSorts		[NsA10]
ns-aterms("LexicalSyntax")	=	scLexicalSyntax		[NsA11]
ns-aterms("ContextFreeSyntax")	=	scContextFreeSyntax		[NsA12]
ns-aterms("Priorities")	=	scPriorities		[NsA13]
ns-aterms("Variables")	=	scVariables		[NsA14]
ns-aterms("Id")	=	scId		[NsA15]
ns-aterms("Sort")	=	scSort		[NsA16]
ns-aterms("LexicalFunction")	=	scLexicalFunction		[NsA17]
ns-aterms("LexIter")	=	scLexIter		[NsA18]
ns-aterms("CharClass")	=	scCharClass		[NsA19]
ns-aterms("Neg")	=	scNeg		[NsA20]
ns-aterms("LexElems")	=	scLexElems		[NsA21]
ns-aterms("QLit")	=	scQLit		[NsA22]
ns-aterms("CfFunction")	=	scCfFunction		[NsA23]
ns-aterms("CfElems")	=	scCfElems		[NsA24]
ns-aterms("Iter")	=	scIter		[NsA25]
ns-aterms("Attributes")	=	scAttributes		[NsA26]
ns-aterms("IncrChain")	=	scIncrChain		[NsA27]
ns-aterms("DecrChain")	=	scDecrChain		[NsA28]
ns-aterms("Abbrev")	=	scAbbrev		[NsA29]
ns-aterms("Abbrevs")	=	scAbbrevs		[NsA30]
ns-aterms("FunctionList")	=	scFunctionList		[NsA31]
ns-aterms("Variable")	=	scVariable		[NsA32]
ns-aterms("SourceEquations")	=	scSourceEquations		[NsA33]
ns-aterms("PrefixEquations")	=	scPrefixEquations		[NsA34]
ns-aterms("When")	=	scWhen		[NsA35]
ns-aterms("Implies")	=	scImplies		[NsA36]
ns-aterms("CondEquation")	=	scCondEquation		[NsA37]
ns-aterms("Condition")	=	scCondition		[NsA38]
ns-aterms("Neq")	=	scNeq		[NsA39]
ns-aterms("Eq")	=	scEq		[NsA40]
ns-aterms("Conditions")	=	scConditions		[NsA41]
ns-aterms("Var")	=	scVar		[NsA42]
ns-aterms("Lex")	=	scLex		[NsA43]
ns-aterms("incremental")	=	scincremental		[NsA44]
ns-aterms(T)	=	T	**otherwise**	[NsA]

aterms(T_1; T_2)	=	aterms(T_1); aterms(T_2)	[AT1]

aterms([T_1 T_2])	=	[aterms(T_1) aterms(T_2)]		[AT2]
aterms(T_1 / T_2)	=	aterms(T_1) / aterms(T_2)		[AT3]
aterms(scModule)	=	`"Module"`		[AT4]
aterms(scSourceSyntax)	=	`"SourceSyntax"`		[AT5]
aterms(scPrefixSyntax)	=	`"PrefixSyntax"`		[AT6]
aterms(scImports)	=	`"Imports"`		[AT7]
aterms(scExports)	=	`"Exports"`		[AT8]
aterms(scHiddens)	=	`"Hiddens"`		[AT9]
aterms(scSorts)	=	`"Sorts"`		[AT10]
aterms(scLexicalSyntax)	=	`"LexicalSyntax"`		[AT11]
aterms(scContextFreeSyntax)	=	`"ContextFreeSyntax"`		[AT12]
aterms(scPriorities)	=	`"Priorities"`		[AT13]
aterms(scVariables)	=	`"Variables"`		[AT14]
aterms(scId)	=	`"Id"`		[AT15]
aterms(scSort)	=	`"Sort"`		[AT16]
aterms(scLexicalFunction)	=	`"LexicalFunction"`		[AT17]
aterms(scLexIter)	=	`"LexIter"`		[AT18]
aterms(scCharClass)	=	`"CharClass"`		[AT19]
aterms(scNeg)	=	`"Neg"`		[AT20]
aterms(scLexElems)	=	`"LexElems"`		[AT21]
aterms(scQLit)	=	`"QLit"`		[AT22]
aterms(scCfFunction)	=	`"CfFunction"`		[AT23]
aterms(scCfElems)	=	`"CfElems"`		[AT24]
aterms(scIter)	=	`"Iter"`		[AT25]
aterms(scAttributes)	=	`"Attributes"`		[AT26]
aterms(scIncrChain)	=	`"IncrChain"`		[AT27]
aterms(scDecrChain)	=	`"DecrChain"`		[AT28]
aterms(scAbbrev)	=	`"Abbrev"`		[AT29]
aterms(scAbbrevs)	=	`"Abbrevs"`		[AT30]
aterms(scFunctionList)	=	`"FunctionList"`		[AT31]
aterms(scVariable)	=	`"Variable"`		[AT32]
aterms(scSourceEquations)	=	`"SourceEquations"`		[AT33]
aterms(scPrefixEquations)	=	`"PrefixEquations"`		[AT34]
aterms(scWhen)	=	`"When"`		[AT35]
aterms(scImplies)	=	`"Implies"`		[AT36]
aterms(scCondEquation)	=	`"CondEquation"`		[AT37]
aterms(scCondition)	=	`"Condition"`		[AT38]
aterms(scNeq)	=	`"Neq"`		[AT39]
aterms(scEq)	=	`"Eq"`		[AT40]
aterms(scConditions)	=	`"Conditions"`		[AT41]
aterms(scVar)	=	`"Var"`		[AT42]
aterms(scLex)	=	`"Lex"`		[AT43]
aterms(T)	=	T	**otherwise**	[AT]

aterms(scincremental) = "incremental" [AT45]

D.2 Is

In the module Is we have defined three boolean functions: isList and isInc. The function isList determines whether an ATerm is a list of ATerms. Note that with the empty list nil as input the function will return false. The function isInc returns true if the input term contains the literal scincremental, otherwise it returns false. The function isInjection returns true if the input represents an injection, otherwise it returns false.

module Is
imports NsATerms$^{D.1}$
exports
context-free syntax
isList(ATerm) → BOOL
isInc(ATerm) → BOOL
isInjection(ATerm) → BOOL
equations
The function isList below returns true if the input ATerm is a list containing two or more elements, otherwise it returns false. Note that although the empty list nil is a list, the application isList(nil) yields false.

isList(T_1; T_2) = true [IL1]
isList(T) = false **otherwise** [IL]

The function isInc below returns true if the Literal scincremental is present as subterm in the input ATerm, otherwise it returns false.

isInc(T) = occurs(scincremental, T) [II1]

The function isInjection below returns true if the input represents an injection, otherwise it returns false.

isInjection([[L [L_1 T]; T_1; T_2] [L_2 L_3; T]]) = true [IInj1]
isInjection(T) = false **otherwise** [IInj]

D.3 MakeNewATerms

In the module MakeNewATerms we have defined some functions that change the structure of an ATerm so that we can use it for various purposes, as for instance error messages or the creation of attribute names.

module MakeNewATerms
imports Integers$^{A.3}$ Is$^{D.2}$
exports
context-free syntax
mk-decl(ATerm) → ATerm
mk-term(ATerm) → ATerm
mk-decls(ATerm) → ATerm
mk-terms(ATerm) → ATerm

cc(ATerm) → ATerm
ccc(ATerm) → ATerm
int2lit(INT) → Literal
lit2int(ATerm) → INT
fun-att(ATerm) → ATerm

variables

Chars [0-9′]∗ → CHAR+

hiddens

context-free syntax

mk-term(ATerm, ATerm) → ATerm

equations

The function mk-decl below expects as input an ATerm representing a context-free function or an AsFixTerm and and produces a Literal that is similar to the declaration of the function, except that eventual spaces and attributes have been removed ("tcp(PROGRAM)→BOOL").

$\text{mk-decl}([\text{scCfFunction } T_1;\ [L\ T_2];\ T_3])$	$= \text{mk-decl}(T_1;\ \text{"->"};\ T_2)$		[MD1]
$\text{mk-decl}([\text{scCfElems } T])$	$= \text{mk-decl}(T)$		[MD2]
$\text{mk-decl}([\text{scSort } T])$	$= T$		[MD3]
$\text{mk-decl}([\text{scQLit } T])$	$= T$		[MD4]
$\text{mk-decl}([\text{scIter } [L_1\ T_1];\ [L_2\ T_2];\ L_3])$	$= \text{mk-decl}(\text{"\{"};\ T_1;\ T_2;\ \text{"\}"};\ L_3)$		[MD5]
$\text{mk-decl}([[T\ T_1]\ T_2])$	$= \text{mk-decl}([T\ T_1])$		[MD6]
$\text{mk-decl}(T_1;\ T_2)$	$= \text{cc}(\text{mk-decl}(T_1);\ \text{mk-decl}(T_2))$		[MD7]
$\text{mk-decl}(L)$	$= L$		[MD8]
$\text{mk-decl}(T)$	$= \text{""}$	**otherwise**	[MD]

The function *mk-term* expects as input an ATerm representing an AsFixTerm and produces a string that is similar to the term it represents.(`"tcp(begin Decls,Series end)"`)

$\text{mk-term}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4])$	$= \text{mk-term}(T_1,\ T_4)$	[MTa]
$\text{mk-term}([\text{scVar } [\text{scQLit } L];\ T_1])$	$= L$	[MTb]
$\text{mk-term}([\text{scVar } L;\ T_2])$	$= L$	[MTc]
$\text{mk-term}([\text{scLex } [\text{scQLit } L];\ T_1])$	$= L$	[MTd]
$\text{mk-term}([\text{scLex } L;\ T_1])$	$= L$	[MTe]

$\text{mk-term}([[\text{scIter } T_1;\ L;\ L_1]\ T_2;\ T_3]) =$ [MTf]
$\text{cc}(\text{mk-term}(T_2);\ L;\ \text{mk-term}([[\text{scIter } T_1;\ L;\ L_1]\ T_3]))$

$\text{mk-term}([[\text{scIter } T_1;\ [\text{scQLit } L];\ L_1]\ T_2;\ T_3]) =$ [MTg]
$\text{cc}(\text{mk-term}(T_2);\ L;\ \text{mk-term}([[\text{scIter } T_1;\ L;\ L_1]\ T_3]))$

$\text{mk-term}([[\text{scIter } T_1;\ L;\ L_1]\ [T_2\ T_3]]) =$ [MTh]
$\text{mk-term}([T_2\ T_3])$

$\text{mk-term}([[\text{scIter } T_1;\ [\text{scQLit } L];\ L_1]\ [T_2\ T_3]]) =$ [MTi]
$\text{mk-term}([T_2\ T_3])$

$\text{mk-term}([[\text{scIter } T]\ \text{nil}]) = \text{nil}$ [MTj]

$\text{mk-term}(T) = \text{nil}$ **otherwise** [MT]

$\text{mk-term}(L;\ T',\ T_1) = \text{cc}(L;\ \text{mk-term}(T',\ T_1))$ [MT1]

$\text{mk-term}(L,\ T_1) = L$ [MT2]

$$\text{mk-term}(T\ /\ T_1,\ T_2) = \text{mk-term}(T,\ T_2) \quad \text{[MT3]}$$
$$\text{mk-term}(T\ /\ T_1;\ T_2,\ T_3) = \text{mk-term}(T;\ T_2,\ T_3) \quad \text{[MT4]}$$

$$\text{mk-term}([[\text{scCfFunction}\ T_1]\ T_2];\ T_3,\ T) = \quad \text{[MT5]}$$
$$\text{cc}(\text{mk-term}([[\text{scCfFunction}\ T_1]\ T_2]);\ \text{mk-term}(T_3,\ T))$$

$$\text{mk-term}([[\text{scCfFunction}\ T_1]\ T_2],\ T) = \text{mk-term}([[\text{scCfFunction}\ T_1]\ T_2]) \quad \text{[MT6]}$$

$$\text{mk-term}([[\text{scIter}\ T_1]\ T_2];\ T_3,\ T) = \quad \text{[MT7]}$$
$$\text{cc}(\text{mk-term}([[\text{scIter}\ T_1]\ T_2]);\ \text{mk-term}(T_3,\ T))$$

$$\text{mk-term}([[\text{scIter}\ T_1]\ T_2],\ T) = \text{mk-term}([[\text{scIter}\ T_1]\ T_2]) \quad \text{[MT8]}$$

$$\text{mk-term}([\text{scVar}\ L_1;\ T_2];\ T',\ T_1) = \text{cc}(L_1;\ \text{mk-term}(T',\ T_1)) \quad \text{[MT9]}$$
$$\text{mk-term}([\text{scVar}\ [L_1\ L_2];\ T_2],\ T_1) = L_2 \quad \text{[MT10]}$$
$$\text{mk-term}([\text{scVar}\ L_1;\ T_2],\ T_1) = L_1 \quad \text{[MT11]}$$

$$\text{mk-term}([\text{scLex}\ L_1;\ T_2];\ T',\ T_1) = \text{cc}(L_1;\ \text{mk-term}(T',\ T_1)) \quad \text{[MT12]}$$
$$\text{mk-term}([\text{scLex}\ [L_1\ L_2];\ T_2],\ T_1) = L_2 \quad \text{[MT13]}$$
$$\text{mk-term}([\text{scLex}\ L_1;\ T_2],\ T_1) = L_1 \quad \text{[MT14]}$$

$$\text{mk-term}([\text{scCfFunction}\ T_1;\ T_2;\ T_3],\ T) = \text{mk-term}(T_1,\ T) \quad \text{[MT15]}$$
$$\text{mk-term}([\text{scCfFunction}\ T_1;\ T_2;\ T_3];\ T_4,\ T) = \text{mk-term}(T_1;\ T_4,\ T) \quad \text{[MT16]}$$

$$\text{mk-term}([\text{scCfElems}\ T_1];\ T_2,\ T) = \text{mk-term}(\text{merge}(T_1,\ T_2),\ T) \quad \text{[MT17]}$$
$$\text{mk-term}([\text{scCfElems}\ T_1],\ T) = \text{mk-term}(T_1,\ T) \quad \text{[MT18]}$$
$$\text{mk-term}([\text{scQLit}\ T_1];\ T_2,\ T) = \text{cc}(T_1;\ \text{mk-term}(T_2,\ T)) \quad \text{[MT19]}$$

$$\text{mk-term}([\text{scQLit}\ T_1],\ T) = T_1 \quad \text{[MT20]}$$
$$\text{mk-term}([\text{scSort}\ T];\ T',\ T_1;\ T_1') = \text{mk-term}(T_1;\ T',\ T_1') \quad \text{[MT21]}$$
$$\text{mk-term}([\text{scSort}\ T],\ T_1;\ T_1') = \text{mk-term}(T_1,\ T_1') \quad \text{[MT22]}$$

$$\text{mk-term}([\text{scSort}\ T];\ T',\ T_1) = \text{mk-term}(T_1;\ T',\ \text{nil}) \quad \textbf{when}\ \text{isList}(T_1) = \text{false} \quad \text{[MT23]}$$
$$\text{mk-term}([\text{scSort}\ T],\ T_1) = \text{mk-term}(T_1,\ \text{nil}) \quad \textbf{when}\ \text{isList}(T_1) = \text{false} \quad \text{[MT24]}$$

The function *mk-decls* takes as input a list of ATerms, representing AsFixTerms and produces a list of Literals representing function declarations, separated by ";".

$$\text{mk-decls}(T;\ T_1) = \text{merge}(\text{mk-decls}(T),\ \text{mk-decls}(T_1)) \quad \text{[MDs1]}$$
$$\text{mk-decls}(\text{nil}) = \text{nil} \quad \text{[MDs2]}$$
$$\text{mk-decls}(T) = \text{mk-decl}(T) \quad \textbf{when}\ \text{isList}(T) = \text{false} \quad \text{[MDs3]}$$

The function *mk-terms* takes as input a list of ATerms, representing AsFixTerms and produces a list of Literals representing function calls, separated by ";".

$$\text{mk-terms}(T;\ T_1) = \text{merge}(\text{mk-terms}(T),\ \text{mk-terms}(T_1)) \quad \text{[MTs1]}$$
$$\text{mk-terms}(\text{nil}) = \text{nil} \quad \text{[MTs2]}$$
$$\text{mk-terms}(T) = \text{mk-term}(T) \quad \textbf{when}\ \text{isList}(T) = \text{false} \quad \text{[MTs3]}$$

The function *cc* takes as argument a number of *Literals* separated by ";" and concatenates them.

$$\mathsf{cc}(L) \ = \ L \qquad \text{[CC1]}$$

$$\mathsf{cc}(\mathsf{literal}(\texttt{"""}\ \mathit{Chars}_1\ \texttt{"""});\, \mathsf{literal}(\texttt{"""}\ \mathit{Chars}_2\ \texttt{"""})) = \mathsf{literal}(\texttt{"""}\ \mathit{Chars}_1\ \mathit{Chars}_2\ \texttt{"""}) \qquad \text{[CC2]}$$

$$\mathsf{cc}(\mathsf{literal}(\texttt{"""}\ \mathit{Chars}_1\ \texttt{"""});\, \mathsf{literal}(\texttt{"""}\ \mathit{Chars}_2\ \texttt{"""});\ T) = \mathsf{cc}(\mathsf{literal}(\texttt{"""}\ \mathit{Chars}_1\ \mathit{Chars}_2\ \texttt{"""});\ T) \qquad \text{[CC3]}$$

The function ccc defined below does the same as the function cc defined above, except that is inserts a comma between the literals.

$$\mathsf{ccc}(L) \ = \ L \qquad \text{[CCC1]}$$
$$\mathsf{ccc}(L_1;\ L_2;\ T) \ = \ \mathsf{ccc}(\mathsf{cc}(L_1;\ \texttt{","};\ L_2);\ T) \qquad \text{[CCC2]}$$
$$\mathsf{ccc}(L_1;\ L_2) \ = \ \mathsf{cc}(L_1;\ \texttt{","};\ L_2) \qquad \text{[CCC3]}$$

The function *int2lit* below takes as argument an integer and converts it into a literal representing that integer.

$$\mathsf{int2lit}(\mathsf{nat\text{-}con}(\mathit{Chars})) \ = \ \mathsf{literal}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""}) \qquad \text{[IL1]}$$

The function lit2int below converts an ATerm into the integer it represents.

$$\mathsf{lit2int}(\mathsf{literal}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""})) \ = \ \mathsf{nat\text{-}con}(\mathit{Chars}) \qquad \text{[LI1]}$$

The function fun-att defined below expects as input an ATerm representing the top operator on the left-hand side of an equation and returns the function attribute associated with this operator.

$$\mathsf{fun\text{-}att}(T) \ = \ \mathsf{cc}(\texttt{"\textbackslash""};\ T;\ \texttt{"\textbackslash""}) \qquad \text{[FA1]}$$

D.4 SimplePrsProjections

In the module SimplePrsProjections we have specified some simple projection functions that select a specific subterm of an ATerm. The names of the functions have be derived from the PRS-theory. The operator is the name of a function. The constructor is the first argument in a function, if this really is a constructor will be checked at a later stage. The parameters are all arguments in a function except the first, the constructor.

module SimplePrsProjections
imports MakeNewATerms$^{D.3}$
exports
  context-free syntax
    operator(ATerm) → ATerm
    constructor(ATerm) → ATerm
    parameters(ATerm) → ATerm
equations

The functions operator, constructor and parameters are defined for ATerms that represent AsFixTerms, thus the left-hand side or the right-hand side of both equations as well as conditions. The function operator returns the ATerm representing the top operator of the AsFixTerm, constructor returns an ATerm representing the constructor function, ie.the first argument of a function and the function parameters the ATerms representing all arguments except the first argument, the constructor.

$$\text{operator}([[\text{scCfFunction } T]\ T_1]) = [\text{scCfFunction } T] \quad [\text{Op1}]$$
$$\text{operator}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{operator}(T_1) \quad [\text{Op2}]$$
$$\text{operator}(T) = \text{nil} \quad \textbf{otherwise} \quad [\text{Op}]$$

$$\text{constructor}([[\text{scCfFunction } T]\ T_1;\ T_2]) = T_1 \quad [\text{Con1}]$$
$$\text{constructor}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{constructor}(T_1) \quad [\text{Con2}]$$

$$\text{constructor}([[\text{scCfFunction } T]\ T_1]) = T_1 \quad [\text{Con3}]$$
when
$$\text{false} = \text{isList}(T_1)$$
$$\text{constructor}(T) = \text{nil} \quad \textbf{otherwise} \quad [\text{Con}]$$

$$\text{parameters}([[\text{scCfFunction } T]\ T_1;\ T_2]) = T_2 \quad [\text{Par1}]$$
$$\text{parameters}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{parameters}(T_1) \quad [\text{Par2}]$$
$$\text{parameters}(T) = \text{nil} \quad \textbf{otherwise} \quad [\text{Par}]$$

D.5 ComplexPrsProjections

In the module ComplexPrsProjections we have specified a number of functions that we use to select information out of a module or a number of modules. This information will be used to check if the specifications specified in the modules are *Well-presented Primitive Recursive Schemes with parameters*, with other words, if they can be transformed into *incremental specifications*. We will also use this function during the transformation of specifications.

module ComplexPrsProjections
imports SimplePrsProjections$^{D.4}$
exports
  context-free syntax
    inc-operators-in-syntax({ATerm ","}*) → ATerm
    inc-equations({ATerm ","}*) → ATerm
    non-inc-equations({ATerm ","}*) → ATerm
    operators-in-all-equations({ATerm ","}*) → ATerm
    vars(ATerm) → ATerm
    inc-operators(ATerm) → ATerm
    operators-in-eqs(ATerm) → ATerm
    inc-terms(ATerm) → ATerm
    inc-constructors(ATerm) → ATerm
    cf-syntax(ATerm) → ATerm
hiddens
  context-free syntax
    inc-eqs(ATerm) → ATerm
    non-inc-eqs(ATerm) → ATerm

equations

The function inc-operators-in-syntax below returns a list of ATerms representing all incremental context-free functions declared in the input modules. Expected input: a list of ATerms representing modules separated by ",".

$$\text{inc-operators-in-syntax}() \ = \ \text{nil} \qquad \text{[Iois1]}$$

$$\text{inc-operators-in-syntax}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) = \qquad \text{[Iois2]}$$
$$\text{inc-operators}(\text{cf-syntax}(T_2))$$

$$\text{inc-operators-in-syntax}(T,\ Ts^+) = \qquad \text{[Iois3]}$$
$$\text{merge}(\text{inc-operators-in-syntax}(T),\ \text{inc-operators-in-syntax}(Ts^+))$$

The function inc-equations below returns a list of ATerms that contain all incremental equations in a module. The ATerms in the list are annotated with the module name. We call an equation incremental if the left-hand side of the equation has an incremental context-free function as as top operator. Expected input: a list of ATerms representing modules separated by ",".

$$\text{inc-equations}() \ = \ \text{nil} \qquad \text{[Ieqs1]}$$
$$\text{inc-equations}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) \ = \ \text{inc-eqs}(T_4)\ /\ L_1 \qquad \text{[Ieqs2]}$$

$$\text{inc-equations}(T,\ Ts^+) = \qquad \text{[Ieqs3]}$$
$$\text{merge}(\text{inc-equations}(T),\ \text{inc-equations}(Ts^+))$$

The function non-inc-equations below is similar to the function inc-equations described above, except that it selects all non incremental equations, ie. having a non-incremental top operator.

$$\text{non-inc-equations}() \ = \ \text{nil} \qquad \text{[Nieqs1]}$$
$$\text{non-inc-equations}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) \ = \ \text{non-inc-eqs}(T_4)\ /\ L_1 \qquad \text{[Nieqs2]}$$

$$\text{non-inc-equations}(T,\ Ts^+) = \qquad \text{[Nieqs3]}$$
$$\text{merge}(\text{non-inc-equations}(T),\ \text{non-inc-equations}(Ts^+))$$

The function all-equations below selects all equations per module an annotates the list of ATerms representing these equations with the name of the module.

$$\text{operators-in-all-equations}() \ = \ \text{nil} \qquad \text{[Oeqs1]}$$

$$\text{operators-in-all-equations}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ [L_3\ \ T_4]]) = \qquad \text{[Oeqs3]}$$
$$\text{operators-in-eqs}(T_4)\ /\ L_1$$

$$\text{operators-in-all-equations}(T,\ Ts^+) = \qquad \text{[Oeqs2]}$$
$$\text{merge}(\text{operators-in-all-equations}(T),\ \text{operators-in-all-equations}(Ts^+))$$

The function vars returns a list of all variable names that are present in the input ATerm.

$$\begin{array}{lll} \text{vars}(T_1;\ T_2) & = \text{merge}(\text{vars}(T_1),\ \text{vars}(T_2)) & & \text{[Var1]} \\ \text{vars}([L\ [L_1\ L_2];\ T_2]) & = L_2 & & \text{[Var2]} \\ \text{vars}([L\ L_1;\ T_2]) & = L_1 & & \text{[Var3]} \\ \text{vars}([T_1\ \ T_2]) & = \text{vars}(T_1;\ T_2) & \textbf{when}\ \ T_1 \neq \text{scVar} & \text{[Var4]} \\ \text{vars}(T) & = \text{nil} & \textbf{otherwise} & \text{[Var]} \end{array}$$

The function inc-operators below selects all operators (or functions) that have been declared as incremental in the input ATerm. This ATerm either represents an equation or is a subterm of an ATerm representing an equation.

$$\text{inc-operators}(T_1;\ T_2) = \text{merge}(\text{inc-operators}(T_1), \text{inc-operators}(T_2)) \quad [\text{Io1}]$$

$$\text{inc-operators}([\text{scCondEquation } L_1;\ L_2;\ T]) = \text{inc-operators}(T) \quad [\text{Io2}]$$

$$\text{inc-operators}([\text{scConditions } T]) = \text{inc-operators}(T) \quad [\text{Io3}]$$

$$\text{inc-operators}([\text{scCondition } L;\ T]) = \text{inc-operators}(T) \quad [\text{Io4}]$$

$$\text{inc-operators}([[\text{scCfFunction } T]\ T_1]) = \text{merge}(\text{inc-operators}([\text{scCfFunction } T]), \text{inc-operators}(T_1)) \quad [\text{Io5}]$$

$$\frac{\text{true} = \text{isInc}(T_3)}{\text{inc-operators}([\text{scCfFunction } T_1;\ T_2;\ T_3]) = [\text{scCfFunction } T_1;\ T_2;\ T_3]} \quad [\text{Io6}]$$

$$\text{inc-operators}(T) = \text{nil} \quad \textbf{otherwise} \quad [\text{Io}]$$

The function operators-in-eqs defined below selects all top operators of the left-hand side of the input equations and annotates them with the tag of the equation.

$$\text{operators-in-eqs}(T_1;\ T_2) = \text{merge}(\text{operators-in-eqs}(T_1), \text{operators-in-eqs}(T_2)) \quad [\text{Oie1}]$$

$$\text{operators-in-eqs}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{operator}(T_1)\ /\ L_2 \quad [\text{Oie2}]$$

The function inc-terms below selects all terms that have been declared as incremental in the input ATerm. This ATerm either represents an equation or is a subterm of an ATerm representing an equation.

$$\text{inc-terms}(T_1;\ T_2) = \text{merge}(\text{inc-terms}(T_1), \text{inc-terms}(T_2)) \quad [\text{It1}]$$

$$\text{inc-terms}([\text{scCondEquation } L_1;\ L_2;\ T]) = \text{inc-terms}(T) \quad [\text{It2}]$$

$$\text{inc-terms}([\text{scConditions } T]) = \text{inc-terms}(T) \quad [\text{It3}]$$

$$\text{inc-terms}([\text{scCondition } T]) = \text{inc-terms}(T) \quad [\text{It4}]$$

$$\frac{\text{true} = \text{isInc}(T_3)}{\text{inc-terms}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ [T_4\ T_5]\ /\ T_6]) = [[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ [T_4\ T_5]\ /\ T_6]} \quad [\text{It5}]$$

$$\frac{\text{true} = \text{isInc}(T_3)}{\text{inc-terms}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ [T_4\ T_5]]) = [[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ [T_4\ T_5]]} \quad [\text{It6}]$$

$$\frac{\text{true} = \text{isInc}(T_3)}{\text{inc-terms}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4;\ T_5]) = \text{merge}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4;\ T_5], \text{inc-terms}(T_5))} \quad [\text{It7}]$$

$$\frac{\text{false} = \text{isInc}(T_3)}{\text{inc-terms}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4]) = \text{inc-terms}(T_4)} \quad [\text{It8}]$$

$$\text{inc-terms}(T) \ = \ \text{nil} \quad \textbf{otherwise} \tag{It}$$

The function inc-constructors selects all constructors of terms that have been declared as incremental in the input ATerm. With the name constructor we mean the first argument of a function. The name constructor has been derived from the PRS-theory.If this first argument really is a constructor will be checked later. The input ATerm either represents an equation or is a subterm of an ATerm representing an equation.

$$\text{inc-constructors}(T_1;\ T_2) = \text{merge}(\text{inc-constructors}(T_1), \text{inc-constructors}(T_2)) \tag{Ic1}$$

$$\text{inc-constructors}([\text{scCondEquation } L_1;\ L_2;\ T]) \ = \ \text{inc-constructors}(T) \tag{Ic2}$$
$$\text{inc-constructors}([\text{scConditions } T]) \ = \ \text{inc-constructors}(T) \tag{Ic3}$$
$$\text{inc-constructors}([\text{scCondition } T]) \ = \ \text{inc-constructors}(T) \tag{Ic4}$$

$$\frac{\text{true} = \text{isInc}(T_3)}{\text{inc-constructors}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4;\ T_5]) = \text{merge}(T_4, \text{inc-constructors}(T_5))} \tag{Ic5}$$

$$\frac{\text{true} = \text{isInc}(T_3),\ \ \text{false} = \text{isList}(T_4)}{\text{inc-constructors}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4]) = T_4} \tag{Ic5}$$

$$\text{inc-constructors}(T) \ = \ \text{nil} \quad \textbf{otherwise} \tag{Ic}$$

The function cf-syntax select all ATerms that represent a context-free syntax section in an ATerm with representing the syntax section of a module in prefix notation or a subterm of such an ATerm.

$$\text{cf-syntax}(T_1;\ T_2) \ = \ \text{merge}(\text{cf-syntax}(T_1), \text{cf-syntax}(T_2)) \tag{Cfs1}$$
$$\text{cf-syntax}([\text{scPrefixSyntax } T]) \ = \ \text{cf-syntax}(T) \tag{Cfs2}$$
$$\text{cf-syntax}([\text{scExports } T]) \ = \ \text{cf-syntax}(T) \tag{Cfs3}$$
$$\text{cf-syntax}([\text{scHiddens } T]) \ = \ \text{cf-syntax}(T) \tag{Cfs4}$$
$$\text{cf-syntax}([\text{scContextFreeSyntax } T]) \ = \ T \tag{Cfs5}$$

$$\text{cf-syntax}(T) \ = \ \text{nil} \quad \textbf{otherwise} \tag{Cfs}$$

The function inc-eqs selects all incremental equations in an ATerm representing the equations in a module in prefix notation.

$$\text{inc-eqs}(T_1;\ T_2) \ = \ \text{merge}(\text{inc-eqs}(T_1), \text{inc-eqs}(T_2)) \tag{Ies1}$$
$$\text{inc-eqs}([\text{scPrefixEquations } T]) \ = \ \text{inc-eqs}(T) \tag{Ies2}$$

$$\frac{\text{true} = \text{isInc}(T)}{\text{inc-eqs}([\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3]) = [\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3]} \tag{Ies3}$$

$$\text{inc-eqs}(T_1) \ = \ \text{nil} \quad \textbf{otherwise} \tag{Ies}$$

The function non-inc-eqs selects all non-incremental equations in an ATerm representing the equations in a module in prefix notation.

$$\text{non-inc-eqs}(T_1;\ T_2) = \text{merge}(\text{non-inc-eqs}(T_1), \text{non-inc-eqs}(T_2)) \tag{Nies1}$$

non-inc-eqs([scPrefixEquations T]) = non-inc-eqs(T) [Nies2]

$$\frac{\text{false} = \text{isInc}(T)}{\text{non-inc-eqs}([\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3]) = [\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3]}$$ [Nies3]

non-inc-eqs(T_1) = nil **otherwise** [Nies]

D.6 Restruct-eqs

In the module Restruct-eqs functions are defined to change the structure of an equation. Variables that are introduced in the conditions (uninstantiated variables) of the equations are removed. The children of the constructor on the left-hand side of the equation are annotated with the rank (child number) of that child. The parameters of function on the left-hand side of the equation that occur in the conditions and in the right-hand side of the equation, are also annotated with the name of the parameter attribute the represent (par(ϕ,j)). When an equation has been restructed, we do not have to take for instance, uninstantiated variables into account, when an equation is processed by another function.

module Restruct-eqs
imports ComplexPrsProjections$^{D.5}$
exports
  context-free syntax
    restruct-eqs(ATerm) → ATerm
    par-att(ATerm, INT) → ATerm
hiddens
  context-free syntax
    restruct(ATerm) → ATerm
    add-info(ATerm, ATerm) → ATerm
    add-info(ATerm, INT, ATerm) → ATerm
    add-par-info(ATerm, ATerm, INT, ATerm) → ATerm
    sub-uninstantiated-vars(ATerm) → ATerm
    sub(ATerm, ATerm) → ATerm
    get-rhs(ATerm) → ATerm
    get-conds(ATerm) → ATerm
equations

The function restruct-eqs defined below handles input that is either a list of equations (the default) or a list of list of equations annotated with the module name. If the input is a list of annotated ATerms, the annotation is restored ([*Reqs*2]).

restruct-eqs(T_1; T_2) = merge(restruct-eqs(T_1), restruct-eqs(T_2)) [Reqs1]
restruct-eqs(T / L) = restruct(T) / L [Reqs2]
restruct-eqs(T) = restruct(T) **otherwise** [Reqs]

The actual restruct function is defined below. First the children of the constructor are annotated with a rank by function add-info. The next step is to annotate all terms on that are defied by the parameters on the left-hand side of the equation. This is done by the function add-par-info. When this has been done, all the uninstantiated variables in the conditions are replaced by their definition in the conditions and the right-hand side of the equation. The uninstatiated variables are substituted by the function sub-uninstantiated-vars. During the restruction of the equation described above, we did not keep the conditions and the right-hand side of the equation separately, thus T_6 in equation $[Req2]$ contains both the conditions and the right-hand side of the equation. The functions get-rhs and get-conds selects, respectively the right-hand side of the equation and the conditions out of T_6.

$$\mathsf{restruct}(T_1;\ T_2) \ = \ \mathsf{merge}(\mathsf{restruct}(T_1),\ \mathsf{restruct}(T_2)) \qquad [\mathsf{Req1}]$$

$$\frac{\begin{array}{l} T_4 = \mathsf{add\text{-}info}(\mathsf{constructor}(T_1),\ T_3;\ T_2), \\ T_5 = \mathsf{add\text{-}par\text{-}info}(\mathsf{mk\text{-}decl}(T_1),\ \mathsf{parameters}(T_1),\ 1,\ T_4), \\ T_6 = \mathsf{sub\text{-}uninstantiated\text{-}vars}(T_5), \\ T_7 = \mathsf{get\text{-}rhs}(T_6), \\ T_8 = [\mathsf{scConditions\ get\text{-}conds}(T_6)] \end{array}}{\begin{array}{l} \mathsf{restruct}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_7;\ T_8] \end{array}} \qquad [\mathsf{Req2}]$$

$$\mathsf{restruct}(T) \ = \ \mathsf{nil} \quad \textbf{otherwise} \qquad [\mathsf{Req}]$$

The function add-inf defined below annotates the children of an constructor with the rank of the child. If an equation contains an injection, the rank 0 is given to the constructor in the conditions or the right-hand side of the equation.

$$\begin{array}{lcll}
\mathsf{add\text{-}info}([T\ T_1],\ T_2) & = & \mathsf{add\text{-}info}(T_1\ /\ \texttt{"0"},\ T_2) & [\mathsf{Addi0}] \\
 & & \textbf{when} & \\
 & & \quad \mathsf{isInjection}([T\ T_1]) = \mathsf{true} & \\
\mathsf{add\text{-}info}([T\ T_1],\ T_2) & = & \mathsf{add\text{-}info}(T_1,\ 1,\ T_2) & [\mathsf{Addi1}] \\
 & & \textbf{when} & \\
 & & \quad \mathsf{isInjection}([T\ T_1]) = \mathsf{false} & \\
\mathsf{add\text{-}info}(T\ /\ T_1,\ T) & = & T\ /\ T_1 & [\mathsf{Addi2}] \\
\mathsf{add\text{-}info}(T\ /\ T_1,\ T_2;\ T_3) & = & \mathsf{merge}(\mathsf{add\text{-}info}(T\ /\ T_1,\ T_2),\ \mathsf{add\text{-}info}(T\ /\ T_1,\ T_3)) & [\mathsf{Addi3}] \\
 & & \textbf{when} & \\
 & & \quad T \neq T_2;\ T_3 & \\
\mathsf{add\text{-}info}(T\ /\ T_1,[T_2\ T_3]) & = & [\mathsf{add\text{-}info}(T\ /\ T_1,\ T_2)\ \mathsf{add\text{-}info}(T\ /\ T_1,\ T_3)] & [\mathsf{Addi4}] \\
 & & \textbf{when} & \\
 & & \quad T \neq [T_2\ T_3] &
\end{array}$$

$$\mathsf{add\text{-}info}(T\ /\ T_1,\ T_2) \ = \ T_2 \quad \textbf{otherwise} \qquad [\mathsf{ai}]$$

$$\begin{array}{lcll}
\mathsf{add\text{-}info}(T_1;\ T_2,\ \mathit{Int},\ T_3) & = & \mathsf{add\text{-}info}(T_2,\ \mathit{Int}+1,\ \mathsf{add\text{-}info}(T_1\ /\ \mathsf{int2lit}(\mathit{Int}),\ T_3)) & [\mathsf{Ai1}] \\
\mathsf{add\text{-}info}([T\ T_1],\ \mathit{Int},\ T_2) & = & \mathsf{add\text{-}info}([T\ T_1]\ /\ \mathsf{int2lit}(\mathit{Int}),\ T_2) & [\mathsf{Ai2}]
\end{array}$$

$$\mathsf{add\text{-}info}(T,\ \mathit{Int},\ T_1) \ = \ T_1 \quad \textbf{otherwise} \qquad [\mathsf{Ai}]$$

Below the function add-par-info is defined. This function annotates all terms in the conditions and the right-hand side of an equation that is defined by a parameter on the left-hand side of the equation with the name of the parameter attribute the parameter represents (par(ϕ,j)).

$$\begin{array}{ll}
\mathsf{add\text{-}par\text{-}info}(T,\ T_1;\ T_2,\ \mathit{Int},\ T_3) = & [\mathsf{APi1}] \\
\quad \mathsf{add\text{-}par\text{-}info}(T,\ T_2,\ \mathit{Int}+1,\ \mathsf{add\text{-}info}(T_1\ /\ \mathsf{par\text{-}att}(T,\ \mathit{Int}),\ T_3)) &
\end{array}$$

$$\mathsf{add\text{-}par\text{-}info}(T,\ [T_1\ \ T_2],\ Int,\ T_3)\ =\ \mathsf{add\text{-}info}([T_1\ \ T_2]\ /\ \mathsf{par\text{-}att}(T,\ Int),\ T_3) \qquad \text{[APi1]}$$
$$\mathsf{add\text{-}par\text{-}info}(T,\ \mathsf{nil},\ Int,\ T_3)\ =\ T_3 \qquad \text{[APi1]}$$

The function sub-uninstantiated-vars is defined below. This function substitutes all uninstatiated (introduced) variables by its definition, the other side of the condition if it is an equality). A variable on the left- or right-hand side of a condition is uninstantiated if it has not been annotated by the functions add-info and add-par-info defined above. These functions should be applied before the function sub-uninstantiated-vars is applied.

$$\mathsf{sub\text{-}uninstantiated\text{-}vars}([\mathsf{scConditions}\ T];\ T_1) = \qquad \text{[Suv1]}$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}(\mathsf{merge}(T,\ T_1))$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}([\mathsf{scCondition}\ \mathsf{scEq};\ [\mathsf{scVar}\ T];\ T_1];\ T_2) = \qquad \text{[Suv2]}$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}(\mathsf{sub}([\mathsf{scVar}\ T]\ /\ T_1,\ T_2))$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}([\mathsf{scCondition}\ \mathsf{scEq};\ T_1;\ [\mathsf{scVar}\ T]];\ T_2) = \qquad \text{[Suv3]}$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}(\mathsf{sub}([\mathsf{scVar}\ T]\ /\ T_1,\ T_2))$$

$$\mathsf{sub\text{-}uninstantiated\text{-}vars}(T_1;\ T_2)\ =\ T_1;\ \mathsf{sub\text{-}uninstantiated\text{-}vars}(T_2)\quad \textbf{otherwise} \qquad \text{[Suv]}$$
$$\mathsf{sub\text{-}uninstantiated\text{-}vars}([T_1\ \ T_2])\ =\ [T_1\ \ T_2]\quad \textbf{otherwise} \qquad \text{[Suv]}$$

The function sub defined below expects two arguments: an annotated ATerm T/T' an a list of other ATerms. All occurences of T in the list of ATerms are substituted by T'.

$$T \neq [T_1\ \ T_2] \Rightarrow \mathsf{sub}(T\ /\ T',\ [T_1\ \ T_2])\ =\ [\mathsf{sub}(T\ /\ T',\ T_1)\ \mathsf{sub}(T\ /\ T',\ T_2)] \qquad \text{[Sub1]}$$
$$T \neq T_1;\ T_2 \Rightarrow\ \mathsf{sub}(T\ /\ T',\ T_1;\ T_2)\ =\ \mathsf{merge}(\mathsf{sub}(T\ /\ T',\ T_1),\ \mathsf{sub}(T\ /\ T',\ T_2)) \qquad \text{[Sub2]}$$

$$\mathsf{sub}(T\ /\ T',\ T)\ =\ T' \qquad \text{[Sub3]}$$
$$\mathsf{sub}(T\ /\ T',\ T_1)\ =\ T_1\quad \textbf{otherwise} \qquad \text{[Sub]}$$

The functions get-rhs and get-conds selects, respectively, the right-hand-side and the conditions of the equation. In the input, the right-hand side of the equation is the last ATerm in the list of ATerms. In case there are no conditions, the input will only contain the right-hand side of the equation.

$$\mathsf{get\text{-}rhs}(T_1;\ T_2)\ =\ \mathsf{get\text{-}rhs}(T_2) \qquad \text{[Grhs1]}$$
$$\mathsf{get\text{-}rhs}([T_1\ \ T_2])\ =\ [T_1\ \ T_2] \qquad \text{[Grhs2]}$$

$$\mathsf{get\text{-}conds}(T;\ [T_1\ \ T_2])\ =\ T \qquad \text{[Gcond1]}$$
$$\mathsf{get\text{-}conds}([T_1\ \ T_2])\ =\ \mathsf{nil} \qquad \text{[Gcond2]}$$
$$\mathsf{get\text{-}conds}(T_1;\ T_2)\ =\ \mathsf{merge}(T_1,\ \mathsf{get\text{-}conds}(T_2))\quad \textbf{otherwise} \qquad \text{[gc]}$$

The function par-att creates a parameter attribute name of the Literal L, that should be Literals representing the function declaration of a function.

$$\mathsf{par\text{-}att}(L,\ Int)\ =\ \mathsf{cc}(\texttt{"\textbackslash""};\ \texttt{"par("};\ L;\ \texttt{","};\ \mathsf{int2lit}(Int);\ \texttt{")"};\ \texttt{"\textbackslash""}) \qquad \text{[PA1]}$$

Appendix E

PRS-check modules

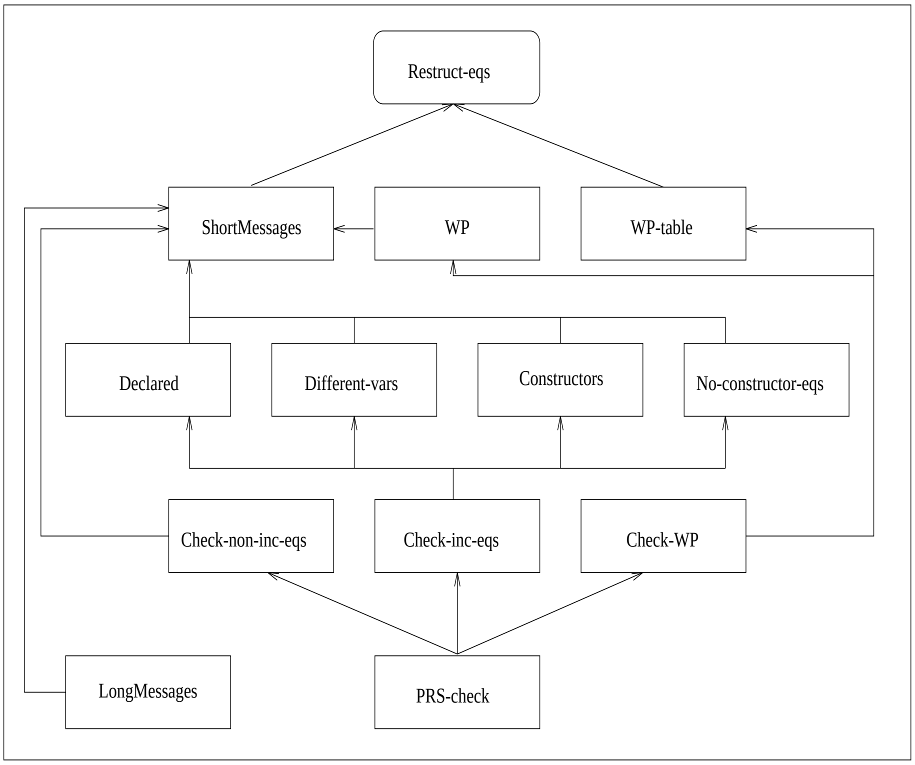


Figure E.1: Module import overview.

E.1 PRS-check

In this module we will check if a specification and all the modules it imports, is a *Primitive recursive scheme with parameters.* A specification can be transformed into an incremental specification if and only if no errors are found here.

module PRS-check
imports Check-non-inc-eqs$^{E.3}$ Check-inc-eqs$^{E.4}$ Check-WP$^{E.5}$
exports
  context-free syntax
    prs-check({ATerm ","}∗, {ATerm ","}∗) → ATerm

equations

The function *prs-check* below expects as input two arguments: the first argument, $Ts+$ is a list of ATerms representing the modules that should be checked, and eventually transformed, and the second argument, Ts contains a list of ATerms that represents all the modules that are imported by the modules in the first argument. Equation [PC2] applies when the list of ATerm representing the modules is empty.

$$\frac{\begin{array}{l} T_1 = \text{inc-operators-in-syntax}(Ts^+), \\ T_2 = \text{inc-equations}(Ts^+), \\ T_3 = \text{non-inc-equations}(Ts^+), \\ T_4 = \text{operators-in-all-equations}(Ts^+, Ts), \\ T_5 = \text{restruct-eqs}(T_2), \\ T_6 = \text{check-non-inc-eqs}(T_3), \\ T_7 = \text{check-inc-eqs}(T_1, T_5, T_4), \\ T_8 = \text{check-wp}(T_5), \\ T_9 = \text{merge}(T_6, \text{merge}(T_7, T_8)) \end{array}}{\text{prs-check}(Ts^+, Ts) = \text{message}(T_9)} \quad \text{[PC1]}$$

$$\text{prs-check}(, Ts) = \text{message}(\text{nil}) \quad \text{[PC2]}$$

E.2 Messages

E.2.1 ShortMessages

In the module ShortMessages we have specified some very short messages that are given to the user concerning the PRS restrictions and the well-presentedness of the specification. Due to the structure of AsFix, it is easy to return Literals as messages and nil if it is not necessary to return a message. The messages generated are very brief. Longer messages can however be generated using the short messages.

module ShortMessages
imports Restruct-eqs$^{D.6}$
exports
  context-free syntax

message(ATerm)	→ ATerm
mod-msg(ATerm)	→ ATerm
eq-msg(ATerm)	→ ATerm
non-inc-eq-msg(ATerm)	→ ATerm
declared-msg(ATerm)	→ ATerm
different-vars-msg(ATerm)	→ ATerm
constructors-msg(ATerm)	→ ATerm
no-default-msg()	→ ATerm
no-list-msg()	→ ATerm
nce-mod-msg(ATerm)	→ ATerm
nce-eq-msg(ATerm)	→ ATerm
no-constructor-eq-msg(ATerm)	→ ATerm
wp-msg(ATerm)	→ ATerm

hiddens
  context-free syntax
    mk-trans-def(ATerm) → ATerm

$$\begin{array}{ll} \mathsf{trans\text{-}def(ATerm)} & \to \mathsf{ATerm} \\ \mathsf{def(ATerm)} & \to \mathsf{ATerm} \\ \mathsf{rmd(ATerm)} & \to \mathsf{ATerm} \end{array}$$

equations

This is the final message. When all the input modules have been tested, the function message will produce the message:`"The module(s) can be transformed"` if no errors are found, otherwise it will give a list of where the errors were found and supply information about the reason for the errors.

$$\mathsf{message(nil)} = \texttt{"The␣module(s)␣can␣be␣transformed"} \quad \text{[M1]}$$

$$\mathsf{message}(T) = \texttt{"The␣module(s)␣cannot␣be␣transformed"};\ T \quad \textbf{when} \quad T \neq \mathsf{nil} \quad \text{[M2]}$$

The function mod-msg expect an annotated ATerm $T\ /\ L$ as input where T is a list errors found in the equations of a module or the empty list nil and L is a module name.

$$\mathsf{mod\text{-}msg}(\mathsf{nil}\ /\ L) = \mathsf{nil} \quad \text{[Modm1]}$$

$$\mathsf{mod\text{-}msg}(T\ /\ L) = \mathsf{cc}(\texttt{"Module␣:␣"};\ L);\ T \quad \textbf{when} \quad T \neq \mathsf{nil} \quad \text{[Modm2]}$$

The function eq-msg expect an annotated ATerm $T\ /\ L$ as input where T is a list errors found in an equation or the empty list nil and L is the tag of the equation.

$$\mathsf{eq\text{-}msg}(\mathsf{nil}\ /\ L) = \mathsf{nil} \quad \text{[Eqm1]}$$

$$\mathsf{eq\text{-}msg}(T\ /\ L) = \mathsf{cc}(\texttt{"␣Eq:["};\ L;\ \texttt{"]␣:"});\ T \quad \textbf{when} \quad T \neq \mathsf{nil} \quad \text{[Eqm2]}$$

The function rmd removes all ATerms that occur more than once.

$$\mathsf{rmd}(T;\ T') = \mathsf{rmd}(T') \quad \textbf{when} \quad \mathsf{occurs}(T,\ T') = \mathsf{true} \quad \text{[RMd1]}$$

$$\mathsf{rmd}(T;\ T') = T;\ \mathsf{rmd}(T') \quad \textbf{when} \quad \mathsf{occurs}(T,\ T') = \mathsf{false} \quad \text{[RMd2]}$$

$$\mathsf{rmd}(L) = L \quad \text{[RMd3]}$$

The specific error messages:

$$\mathsf{non\text{-}inc\text{-}eq\text{-}msg(nil)} = \mathsf{nil} \quad \text{[NIEqm1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{non\text{-}inc\text{-}eq\text{-}msg}(T\ /\ L) = \\ (\texttt{"Incremental␣terms␣in␣equation␣with␣non-inc␣lhs␣:"}; \\ \ \mathsf{ccc}(T))\ /\ L\end{array}} \quad \text{[NIEqm2]}$$

$$\mathsf{declared\text{-}msg(nil)} = \mathsf{nil} \quad \text{[Declm1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{declared\text{-}msg}(T) = \\ \texttt{"Incremental␣functions␣not␣declared␣in␣checked␣specification␣:"}; \\ \mathsf{ccc}(\mathsf{rmd}(T))\end{array}} \quad \text{[Declm2]}$$

$$\mathsf{different\text{-}vars\text{-}msg(nil)} = \mathsf{nil} \quad \text{[DVarm1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{different\text{-}vars\text{-}msg}(T) = \\ \texttt{"Variables␣that␣occur␣more␣than␣once␣on␣lhs␣of␣equation␣:␣"};\\ \mathsf{ccc}(\mathsf{rmd}(T))\end{array}} \quad \text{[Dvarm2]}$$

$$\mathsf{constructors\text{-}msg}(\mathsf{nil}) = \mathsf{nil} \quad \text{[Conm1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{constructors\text{-}msg}(T) = \\ \texttt{"First␣argument␣no␣subterm␣of␣first␣argument␣lhs␣equation␣:␣"};\\ \mathsf{ccc}(T)\end{array}} \quad \text{[Conm2]}$$

$$\begin{array}{l}\mathsf{no\text{-}default\text{-}msg}\ (\ ) = \\ \texttt{"Sorry,␣we␣do␣not␣allow␣any␣default␣equations"}\end{array} \quad \text{[NDM]}$$

$$\begin{array}{l}\mathsf{no\text{-}list\text{-}msg}\ (\ ) = \\ \texttt{"First␣argument␣lhs␣equation␣defined␣as␣list,␣not␣yet␣implemented"}\end{array} \quad \text{[NLM]}$$

$$\mathsf{nce\text{-}mod\text{-}msg}(\mathsf{nil}\ /\ L) = \mathsf{nil} \quad \text{[NModm1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{nce\text{-}mod\text{-}msg}(T\ /\ L) = \\ \mathsf{cc}(\texttt{"␣In␣module␣:␣"}; L); \mathsf{ccc}(T)\end{array}} \quad \text{[NModm2]}$$

$$\mathsf{nce\text{-}eq\text{-}msg}(L) = \mathsf{cc}(\texttt{"Eq:["}; L; \texttt{"]"}) \quad \text{[NEqm1]}$$

$$\mathsf{no\text{-}constructor\text{-}eq\text{-}msg}(\mathsf{nil}\ /\ L) = \mathsf{nil} \quad \text{[NoConEq1]}$$

$$\frac{T \neq \mathsf{nil}}{\begin{array}{l}\mathsf{no\text{-}constructor\text{-}eq\text{-}msg}(T\ /\ L) = \\ \texttt{"Equations␣exists␣for␣first␣argument␣of␣"}; L;\\ \hfill T\end{array}} \quad \text{[NoConEq2]}$$

The function wp-msg is used to create an error message when the specification is not well-presented. The functions wp-aux, wp-aux1 and rmd are auxiliary functions to make the information easier to understand for the user.

$$\mathsf{wp\text{-}msg}(\mathsf{nil}) = \mathsf{nil} \quad \text{[WPm1]}$$

$$\begin{array}{l}\mathsf{wp\text{-}msg}(T) = \\ \texttt{"WP-error,␣some␣parameter-terms␣are␣not␣uniquely␣defined"};\\ \mathsf{rmd}(\mathsf{mk\text{-}trans\text{-}def}(T))\end{array} \quad \text{[WPm2]}$$

The function mk-trans-def creates the transitive definition given the information concerning the definition of a parameter by means of function trans-def.

$$\mathsf{mk\text{-}trans\text{-}def}(T_1;\ T_2) = \mathsf{merge}(\mathsf{mk\text{-}trans\text{-}def}(T_1), \mathsf{mk\text{-}trans\text{-}def}(T_2)) \quad \text{[MTDef1]}$$

$$\mathsf{mk\text{-}trans\text{-}def}([T_1\ \ T_2]) = \mathsf{trans\text{-}def}(T_1); \mathsf{trans\text{-}def}(T_2) \quad \text{[MTDef1]}$$

The function trans-def defined below creates a string of the information, the tag, the term and the parameter by means of the function def and puts the key word "defined by" between these strings.

trans-def(T_1; T_2) = def(T_1); "defined␣by"; trans-def(T_2) [TDef1]
trans-def(T) = def(T) **otherwise** [TDef]

The function def defined below concatenates the tag of the equation L_1, the string representing the term L_2 and string representing the parameter L_3.

def(L_1 / L_2 / L_3) = cc("Eq:["; L_1; "]␣"; L_2; "␣Parameter:␣"; L_3) [Def1]
def(L_1 / L_2) = cc("Eq:["; L_1; "]␣"; L_2) [Def2]
def(L) = L [Def3]

E.2.2 LongMessages

In the module LongMessages we have defined three functions: long-msg, add-long-msg. and get-long-msgs. The function long-msg expects as input a Literal representing a short-message. These short messages are defined in the module ShortMessages. If long-msg is called with a short message a longer message is selected, that gives a longer and more detailed explanation than the short message. The function long-msg can be used interactively. If for instance SEAL is used and a short message is written in a term editor, a SEAL-script can be written to collect the long messages and expose them in another term editor, if the focus is set on a short message.

The function add-long-msgs expects an ATerm representing an error message as input. For all short messages in the input message, the corresponding long messages is added at the beginning of the message. If a user does not understand the short message, he will be able to look at the beginning of the message to find the longer explanations.

The function get-long-msgs returns a list of all long messages corresponding to the short messages in the input that represents a message. This function can be used if it is desirable to put all the longer messages in a separate file or window.

module LongMessages
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    long-msg(ATerm) → ATerm
    add-long-msgs(ATerm) → ATerm
    get-long-msgs(ATerm) → ATerm
hiddens
  context-free syntax
    long-msgs(ATerm) → ATerm
    rm(ATerm, ATerm) → ATerm
equations

long-msg("Incremental␣terms␣in␣equation␣with␣non-inc␣lhs␣:") = [LM1]
"␣The␣terms␣listed␣have␣incremental␣top␣operators␣but␣occur␣in␣␣";
"␣an␣equation␣which␣left-hand␣side␣has␣a␣non␣incremental␣␣␣␣␣␣␣␣";
"␣operator␣␣"

long-msg("Incremental␣functions␣not␣declared␣in␣checked␣specification␣:") = [LM2]

"␣The␣functions␣listed␣are␣not␣declared␣in␣the␣modules␣that␣␣␣␣␣";
"␣are␣to␣be␣transformed.␣This␣would␣mean␣that␣the␣equations␣␣␣␣␣";
"␣containing␣these␣function␣applications␣would␣be␣transformed␣␣␣";
"␣but␣the␣function␣declaration␣would␣not:␣All␣incremental␣␣␣␣␣␣␣";
"␣functions␣must␣be␣declared␣in␣the␣modules␣you␣transform.␣␣␣␣␣␣"

long-msg("Variables␣that␣occur␣more␣than␣once␣on␣lhs␣of␣equation␣:␣") = [LM3]
"␣The␣variables␣listed␣occur␣more␣than␣once␣in␣the␣left-hand␣␣␣␣";
"␣side␣of␣the␣equation.␣If␣you␣want␣to␣be␣sure␣that␣two␣terms␣␣␣";
"␣are␣equal␣in␣the␣lhs␣of␣the␣equation,␣use␣variables␣with␣␣␣␣␣␣";
"␣different␣names␣and␣introduce␣conditions␣to␣check␣equality␣␣␣␣"

long-msg("First␣argument␣no␣subterm␣of␣first␣argument␣lhs␣equation␣:␣") = [LM4]
"␣The␣first␣variable␣in␣the␣listed␣terms␣do␣not␣represent␣␣␣␣␣␣␣";
"␣a␣subterm␣of␣the␣first␣argument␣on␣the␣left-hand␣side␣of␣␣␣␣␣␣";
"␣the␣equation.␣Make␣sure␣that␣it␣is␣or␣remove␣the␣attribute␣␣␣␣";
"␣incremental␣in␣the␣declarations␣of␣the␣functions␣listed.␣␣␣␣␣␣"

long-msg("Equations␣exists␣for␣first␣argument␣of␣") = [LM5]
"␣The␣first␣argument␣in␣this␣function␣should␣be␣a␣constructor␣␣␣";
"␣but␣an␣equation␣for␣this␣term␣has␣been␣found␣and␣therefore␣it␣";
"␣is␣not␣a␣constructor␣term.␣You␣can␣remove␣the␣equations␣listed";
"␣if␣it␣does␣not␣affect␣your␣program␣or␣remove␣the␣attribute␣␣␣␣";
"␣incremental␣in␣the␣declaration␣of␣this␣function.␣␣␣␣␣␣␣␣␣␣␣␣␣␣"

long-msg("WP-error,␣some␣parameter-terms␣are␣not␣uniquely␣defined") = [LM6]
"␣The␣left-most␣parameter␣terms␣are␣defined␣by␣different␣terms.␣";
"␣The␣right-most␣terms␣in␣the␣transitive␣definition␣given␣defines";
"␣the␣value␣of␣the␣left-most␣term.␣See␣if␣you␣can␣change␣the␣␣␣␣";
"␣equations␣or␣simply␣make␣the␣functions␣non-incremental␣by␣␣␣␣␣";
"␣removing␣the␣attribute␣incremental␣in␣the␣declarations␣given.␣";
"␣There␣are␣no␣real␣constructive␣advice␣to␣give.␣SORRY␣␣␣␣␣␣␣␣␣␣␣"

long-msg(T) = "No␣explanation" **otherwise** [LM]

add-long-msgs(T) = merge(get-long-msgs(T), T) [AL1]

$T \neq$ "The␣module(s)␣can␣be␣transformed"
―――――――――――――――――――――――――――――― [GL1]
get-long-msgs(T) =
"Detailed␣explanations␣:␣";
long-msgs(T)

get-long-msgs("The␣module(s)␣can␣be␣transformed") = [GL2]
"No␣long␣messages␣needed"

Function long-msg defined below: If the long message corresponding to T_1 is "No explanation" T_2 is searched for messages corresponding to the long messages. If the long message corresponding to T_1 is not equal to "No explanation", a short message has been found. T_1 and T are merged with the result of searching the rest of the list of messages, after that all occurences of T_1 in this list have been removed; a long message is only needed once.

$$\frac{\begin{array}{l} T = \mathsf{long\text{-}msg}(T_1), \\ T \neq \texttt{"No␣explanation"} \end{array}}{\mathsf{long\text{-}msgs}(T_1;\ T_2) = \mathsf{merge}(T_1;\ T, \mathsf{long\text{-}msgs}(\mathsf{rm}(T_1,\ T_2)))} \quad \text{[LS1]}$$

$$\frac{\texttt{"No␣explanation"} = \mathsf{long\text{-}msg}(T_1)}{\mathsf{long\text{-}msgs}(T_1;\ T_2) = \mathsf{long\text{-}msgs}(T_2)} \quad \text{[LS1]}$$

$$\mathsf{long\text{-}msgs}(T) \ = \ \mathsf{nil} \quad \textbf{otherwise} \quad \text{[LS]}$$

Remove messages for which we have already found a corresponding long message.

$$\begin{array}{lcll} \mathsf{rm}(L, L;\ T) & = & \mathsf{rm}(L,\ T) & & \text{[RM1]} \\ \mathsf{rm}(L, L_1;\ T) & = & \mathsf{merge}(L_1, \mathsf{rm}(L,\ T)) & \textbf{when } L \neq L_1 & \text{[RM2]} \\ \mathsf{rm}(L, L) & = & \mathsf{nil} & & \text{[RM3]} \\ \mathsf{rm}(L, L_1) & = & L_1 & \textbf{when } L \neq L_1 & \text{[RM4]} \end{array}$$

E.3 Check-non-inc-eqs

In this module all equations with a top-operator that is not declared as incremental are checked so that no other operators in the equations are declared as incremental. If equations are found with incremental operators in the right-hand side or in the conditions of the equation, an error message will be the result. This error message contains the name of the module, equation tag and the name of the functions involved.

module Check-non-inc-eqs
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    check-non-inc-eqs(ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    check-modules(ATerm) $\rightarrow$ ATerm
    check-equations(ATerm) $\rightarrow$ ATerm
equations
The function check-non-inc-eqs below expects as input a list of annotated ATerms T/L. T contains the non-incremental equations the module L.

$$\mathsf{check\text{-}non\text{-}inc\text{-}eqs}(T) \ = \ \mathsf{check\text{-}modules}(T) \quad \text{[Cnie]}$$

The function check-modules below checks the non-incremental equations per module. Given the result of checking the equations, the function mod-msg is applied. If the result of checking the equations is nil, the function mod-msg returns nil, otherwise an error message is returned. This message starts with the name of the module in which the errors where found, followed by the information returned by the function check-equations. The name of the module is represented by the L in the equation $[Cmod2]$ below.

$$\begin{array}{lcll} \mathsf{check\text{-}modules}(T_1;\ T_2) & = & \mathsf{merge}(\mathsf{check\text{-}modules}(T_1), \mathsf{check\text{-}modules}(T_2)) & \text{[Cmod1]} \\ \mathsf{check\text{-}modules}(T\ /\ L) & = & \mathsf{mod\text{-}msg}(\mathsf{check\text{-}equations}(T)\ /\ L) & \text{[Cmod2]} \end{array}$$

The function check-equations defined below checks the non-incremental equations individually. This is done by first applying the function inc-terms. This function returns all terms in the equations that have been declared as incremental or nil if no incremental terms where found. The function mk-terms returns strings that look like the terms in ASF+SDF-notation. If no errors are found in an equation the function check-equations returns nil, otherwise it returns an error message per equation indicating which terms are concerned. L_2 in equation $[Ceq2]$ below represents the tag of the equation.

check-equations(T_1; T_2) = merge(check-equations(T_1), check-equations(T_2)) [Ceq1]

check-equations([scCondEquation L_1; L_2; T]) = [Ceq2]
eq-msg(non-inc-eq-msg(mk-terms(inc-terms(T)) / L_2))

check-equations(T) = nil **otherwise** [Ceq]

E.4 Check-inc-eqs

In this module all equations with a top-operator that is declared as incremental are checked.

module Check-inc-eqs
imports Declared$^{E.4.1}$ Different-vars$^{E.4.2}$ Constructors$^{E.4.3}$ No-constructor-eqs$^{E.4.4}$
exports
  context-free syntax
    check-inc-eqs(ATerm, ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    check-modules(ATerm, ATerm, ATerm) → ATerm
    check-equations(ATerm, ATerm, ATerm) → ATerm

    no-default(ATerm, ATerm) → ATerm
    no-list(ATerm) → ATerm
  variables
    Chars [0-9]∗ → CHAR∗
equations

The function check-inc-eqs defined below checks all the incremental equations. In equation $[Cie1]$, T_1 contains the declarations of the incremental functions that have been found in the syntax sections, T_2 the incremental equations we want to check and T_3 contains the top operators of all equations that are in the modules we are checking and in the modules imported by these modules.

check-inc-eqs(T_1, T_2, T_3) = check-modules(T_1, T_2, T_3) [Cie1]

The function check-modules defined below invokes the check on the incremental equations per module. This enables us to report errors per module. If no errors are found in a module nil is returned, otherwise a message is returned with the name of the module followed by a list of errors that where found in that module.

check-modules(T_1, T_2; $T_2{}'$, T_3) = [Cmod1]
merge(check-modules(T_1, T_2, T_3), check-modules(T_1, $T_2{}'$, T_3))
check-modules(T_1, T_2 / L, T_3) = [Cmod2]
mod-msg(check-equations(T_1, T_2, T_3) / L)

The function check-equations checks the incremental equations individually, enabling us to report errors found per equation. An incremental equation must satisfy six requirements. The first is that all incremental terms in the equation have been declared in the modules we are checking. This check is performed when the function declared is applied. The next check is to make sure that the equation is left-linear: all variables in the left hand side of the equation have different names. This is done by the function different-vars. The third property we check is that the constructors of the incremental terms in the conditions and on the right-hand side of an incremental equation are either children or equal to the constructor on the left-hand side of the equation. This check is performed by applying the function constructors. The next check assures us that there are no equations defined for the constructor on the left-hand side of this equation. This check is performed by the function no-constructor-eqs. We do not allow equations defining incremental functions to be default equations. This check is performed when the function no-default is applied. The method for incremental rewriting with incremental specifications does not (yet) cover constructors that are defined as list functions. The last check returns a warning if this is the case.

$$\text{check-equations}(T_1,\ T_2;\ T_2{}',\ T_3) = \qquad \text{[Ceq1]}$$
$$\text{merge}(\text{check-equations}(T_1,\ T_2,\ T_3),\ \text{check-equations}(T_1,\ T_2{}',\ T_3))$$

$$\frac{\begin{array}{l} T_4 = \text{declared}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T), \\ T_5 = \text{different-vars}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]), \\ T_6 = \text{constructors}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]), \\ T_7 = \text{no-constructor-eqs}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T'), \\ T_8 = \text{no-default}(L_2,\ \texttt{"default"}), \\ T_9 = \text{no-list}(\text{constructor}(T_1)), \\ T_{10} = \text{merge}(\text{merge}(T_4,\ T_5),\ \text{merge}(\text{merge}(T_6,\ T_7),\ \text{merge}(T_8,\ T_9))) \end{array}}{\begin{array}{l} \text{check-equations}(T,\ [\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T') = \\ \text{eq-msg}(T_{10}\ /\ L_2) \end{array}} \qquad \text{[Ceq2]}$$

$$\text{check-equations}(T_1,\ T_2,\ T_3)\ =\ \text{nil} \quad \textbf{otherwise} \qquad \text{[Ceq]}$$

The function no-default defined below returns nil if the tag, the first argument is not a default equation tag, otherwise an error message.

$$\text{no-default}(\text{literal}(\texttt{"\""}\ \textit{Chars}\ \textit{Chars}_1\ \texttt{"\""}),\ \text{literal}(\texttt{"\""}\ \textit{Chars}\ \texttt{"\""})) = \qquad \text{[Ndef1]}$$
$$\text{no-default-msg}\ (\ )$$
$$\text{no-default}(\text{literal}(\texttt{"\""}\ \textit{Chars}\ \texttt{"\""}),\ \text{literal}(\texttt{"\""}\ \textit{Chars}\ \texttt{"\""})) = \qquad \text{[Ndef2]}$$
$$\text{no-default-msg}\ (\ )$$

$$\text{no-default}(T,\ T')\ =\ \text{nil} \quad \textbf{otherwise} \qquad \text{[Ndef]}$$

The function no-list defined below returns nil if the constructor, the input, is not a list construction, otherwise it returns an error message.

$$\begin{array}{lll} \text{no-list}(T) & = & \text{no-list-msg}\ (\ ) \qquad \text{[Nlist1]} \\ & & \textbf{when} \\ & & \quad \text{true} = \text{occurs}(\text{scIter},\ T) \\ \text{no-list}(T) & = & \text{nil} \qquad \textbf{otherwise} \qquad \text{[Nlist]} \end{array}$$

E.4.1 Declared

In this module we will check that all incremental operators occurring in an incremental equation is defined in the modules we are checking, and want to transform. An incremental operator must be declared in these modules.

module Declared
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    declared(ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    check-operators(ATerm, ATerm) $\rightarrow$ ATerm
equations

The function declared defined below checks that all incremental functions occuring in an equation have been declared in the modules that are checked. T contains the declarations of all these functions. All incremental terms in the equation are selected by means of function inc-terms and the function check-operators is applied. The function check-operators returns all terms which top operators are not present in T. If all these top operators are present they have all been declared and the function check-operators returns nil.

$$\text{declared}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T) = \text{declared-msg}(\text{mk-terms}(\text{check-operators}(\text{inc-terms}(T_1;\ T_2;\ T_3),\ T))) \quad [\text{D}]$$

$$\text{check-operators}(T;\ T',\ T_1) = \text{merge}(\text{check-operators}(T,\ T_1),\ \text{check-operators}(T',\ T_1)) \quad [\text{Co1}]$$

$$\frac{\text{false} = \text{occurs}([\text{scCfFunction } T],\ T_1)}{\text{check-operators}([[\text{scCfFunction } T]\ T'],\ T_1) = [[\text{scCfFunction } T]\ T']} \quad [\text{Co2}]$$

$$\text{check-operators}(T,\ T_1)\ =\ \text{nil} \quad \textbf{otherwise} \quad [\text{Co}]$$

E.4.2 Different-vars

In this module the check on the variables in an incremental equation is specified. The top operator of an incremental equation may not contain the same variable more than once.

module Different-vars
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    different-vars(ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    check-variables(ATerm) $\rightarrow$ ATerm
equations

The function different-vars below expects as input an ATerm representing an incremental equation. The variables in the left-hand side of the equation, represented by T_1, are selected by means of the function vars that returnes all variables present in the input. To these variables the function check-variables is applied. The function different-vars returns a list of literals representing all variables that occurred more than once on the left-hand side of the equation. If no variables occur more than once, nil will be returned as result of the function.

$$\text{different-vars}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{different-vars-msg}(\text{check-variables}(\text{vars}(T_1))) \quad [\text{Dv}]$$

The function check-variables below recursively checks if the head of the input list is present in the tail. If the head, a variable, is present in the tail it is returned, together with the result of checking the rest of the list. If no variable occurs more than once nil is returned.

$$\frac{\mathsf{true} = \mathsf{occurs}(T_1,\ T_2)}{\mathsf{check\text{-}variables}(T_1;\ T_2) = \mathsf{merge}(T_1, \mathsf{check\text{-}variables}(T_2))} \quad [\mathsf{Cv1}]$$

$$\frac{\mathsf{false} = \mathsf{occurs}(T_1,\ T_2)}{\mathsf{check\text{-}variables}(T_1;\ T_2) = \mathsf{check\text{-}variables}(T_2)} \quad [\mathsf{Cv2}]$$

$$\mathsf{check\text{-}variables}(T) \ = \ \mathsf{nil} \quad \textbf{otherwise} \quad [\mathsf{Cv}]$$

E.4.3 Constructors

In this module we check the first argument of all incremental functions in an incremental equations. All these first arguments must be a subterm of the grammar term in the top operator.

module Constructors
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    constructors(ATerm) → ATerm
hiddens
  context-free syntax
    check-constructors(ATerm) → ATerm
equations

The function constructors defined below takes an ATerm representing an incremental equation as argument. A check is performed to make sure that the first argument of all terms in the equation (except the top operator of the left-hand side of the equation) is a child of the constructor on the left-hand side of the equation. All incremental terms are selected in the conditions and the right-hand side of the equation by means of function inc-terms. With these incremental terms the function check-constructors is applied.

$$\frac{T = \mathsf{inc\text{-}terms}(\mathsf{parameters}(T_1);\ T_2;\ T_3)}{\begin{array}{l}\mathsf{constructors}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \\ \mathsf{constructors\text{-}msg}(\mathsf{check\text{-}constructors}(T))\end{array}} \quad [\mathsf{Co}]$$

The function *check-constructors* below checks that the constructors of the functions have been annotated. Prior to this check the function restruct-eqs should have been applied wherein all children of a constructor are annotated with a rank. If this rank is not present we know that the constructor of the term is not a child of the constructor of the left-hand side of the equation. If no annotation is found, a string is returned representing the term, otherwise the function yields nil.

$$\mathsf{check\text{-}constructors}(T_1;\ T_2) \ = \ \mathsf{merge}(\mathsf{check\text{-}constructors}(T_1), \mathsf{check\text{-}constructors}(T_2)) \quad [\mathsf{Cco1}]$$

$$\mathsf{check\text{-}constructors}([[\mathsf{scCfFunction}\ T]\ T_1\ /\ T_2;\ T_3]) \ = \ \mathsf{nil} \quad [\mathsf{Cco2}]$$

$$\mathsf{check\text{-}constructors}([[\mathsf{scCfFunction}\ T]\ T_1\ /\ T_2]) \ = \ \mathsf{nil} \quad [\mathsf{Cco3}]$$

$$\mathsf{check\text{-}constructors}(T) \ = \ \mathsf{mk\text{-}terms}(T) \quad \textbf{otherwise} \quad [\mathsf{Cco}]$$

E.4.4 No-constructor-eqs

In this module we will check that there are no equations, or rewrite rules if you wish, defined for the constructors of terms defined as incremental. If a function has been declared as incremental, its first argument should be a constructor and no rewrite rules may exist over these constructors.

module No-constructor-eqs
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    no-constructor-eqs(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    check-modules(ATerm, ATerm) → ATerm
    check-equations(ATerm, ATerm) → ATerm
equations

We do not allow equations defined for the constructors. The function *no-constructor-eqs* below checks if there are any equations defined for the constructors in a specification. In equation $[NCe]$ given below T_4 is the constructor of a function and T_5 represents the constructors variables. T is a list of ATerms, $(T_i/L_i; \ldots T_n/L_n)/L$ where T_j is the top operator in an equations and L_j is the tag of that equation. L is the name of the module in which these operators and tags occur. We will check that T_4, the constructor of this equation does not occur in the lists of operators by means of the function check-modules.

$$\frac{[T_4\ T_5] = \text{constructor}(T_1)}{\begin{array}{l}\text{no-constructor-eqs}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T) = \\ \text{no-constructor-eq-msg}(\text{check-modules}(T_4,\ T)\ /\ \text{mk-term}(T_1))\end{array}} \quad [\text{NCe}]$$

The function check-modules enables us to inform the user in which module an eventual equation is defined for the constructor (*nce* is *no constructor equations* abbreviated).

$$\text{check-modules}(T,\ T_1;\ T_2) = \text{merge}(\text{check-modules}(T,\ T_1), \text{check-modules}(T,\ T_2)) \quad [\text{Cmod1}]$$

$$\text{check-modules}(T,\ T_1\ /\ L) = \text{nce-mod-msg}(\text{check-equations}(T,\ T_1)\ /\ L) \quad [\text{Cmod2}]$$

The function *check-equations* below returns the tags of all equation of which the top operator equal is to the constructor.

$$\begin{array}{l}\text{check-equations}(T,\ T_1;\ T_2) = \\ \ \text{merge}(\text{check-equations}(T,\ T_1), \text{check-equations}(T,\ T_2))\end{array} \quad [\text{Ceq1}]$$

$$\text{check-equations}(T,\ T\ /\ L) = \text{nce-eq-msg}(L) \quad [\text{Ceq2}]$$

$$\text{check-equations}(T,\ T_1) = \text{nil} \qquad \textbf{otherwise} \quad [\text{Ceq}]$$

E.5 Check-WP

In the module Check-WP the test for checking whether a specification is well-presended or not. The function check-wp defined in this module first selects information about how parameter terms in the specification are defined by calling the function wp-table that returns a table containing the definition rules of the parameter terms. With three copies of this table, the function wp is called. This function yields nil if all parameter terms are uniquely defined and information for producing an error message otherwise.

module Check-WP
imports WP[E.5.2] WP-table[E.5.1]
exports
  context-free syntax
    check-wp(ATerm) → ATerm
equations

$$\frac{\begin{array}{c} T' = \text{wp-table}(T), \\ T_1 = \text{wp}(T',\ T',\ T') \end{array}}{\text{check-wp}(T) = \text{wp-msg}(T_1)} \quad \text{[Cwp1]}$$

E.5.1 WP-table

In the module WP-table the definitions of the functions we use to build the table we use to check the (PRS) well-presentedness of a specification. The table produced contains the definition rules of all parameter terms in the specification.

module WP-table
imports Restruct-eqs[D.6]
exports
  context-free syntax
    wp-table(ATerm) → ATerm
hiddens
  context-free syntax
    table-per-equation(ATerm) → ATerm
    table-per-term(ATerm, ATerm, ATerm) → ATerm
    table-per-parameter(ATerm, ATerm, ATerm, ATerm, ATerm, INT) → ATerm
equations

The function wp-table constructs a table we will use when we check whether all parameter attributes are uniquely defined. This table contains information about all parameter attributes in the specification and their definition.

$$\text{wp-table}(T_1;\ T_2) = \text{merge}(\text{wp-table}(T_1), \text{wp-table}(T_2)) \quad \text{[WPt1]}$$

$$\text{wp-table}(T\ /\ T_1) = \text{wp-table}(T) \quad \text{[WPt2]}$$

$$\text{wp-table}(T) = \text{table-per-equation}(T) \quad \textbf{otherwise} \quad \text{[WPt]}$$

table-per-equation: For every equation in the input the following actions are taken. The constructor without variables is selected on the left-hand side of the equation by means of the function constructor. This constructor is represented by the variable T_4 in equation [$Tpe2$]. The function inc-terms selects all incremental terms in the conditions and on the right-hand side of the equation, T_6. With T_4, T_6 and the tag of the equation, L_2 as arguments, the function table-per-term is invoked.

$$\text{table-per-equation}(T_1;\ T_2) = \quad \text{[Tpe1]}$$
$$\text{merge}(\text{table-per-equation}(T_1), \text{table-per-equation}(T_2))$$

$$\frac{\begin{array}{c} [T_4\ \ T_5] = \text{constructor}(T_1), \\ T_6 = \text{inc-terms}(T_2;\ T_3) \end{array}}{\begin{array}{l} \text{table-per-equation}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \\ \text{table-per-term}(L_2,\ T_4,\ T_6) \end{array}} \quad \text{[Tpe2]}$$

table-per-term: For every incremental term the following is done. If the constructor in the term is annotated (T_3/L_3), L_3 is the rank of the constructor of the term, [*Tpt*2], a declaration representation ("`tcs(SERIES,TEN V)->BOOL`") is made of the term by means of function mk-decl (T_5) and a term representation
("`tcs(Series,Tenv)`") is made of the same term by means of the function mk-term (T_6). The function table-per-parameter will now be invoked with the following arguments. The tag of the equation, L, the constructor of the left-hand side of the equation annotated with the rank of the constructor of the term T_1/L_3, the declaration representation of the term T_5, the term representation of the term T_6, the parameters of the term T_4 and a counter "1"; for each parameter in the term an entry in the table must be made. This counter will be used to numerate the corresponding parameter attributes.

$$\text{table-per-term}(L,\ T_1,\ T_2;\ T_3) = \qquad [\text{Tpt1}]$$
$$\text{merge}(\text{table-per-term}(L,\ T_1,\ T_2),\ \text{table-per-term}(L,\ T_1,\ T_3))$$

$$\frac{\begin{array}{l} T_5 = \text{mk-decl}([\text{scCfFunction}\ T_2]), \\ T_6 = \text{mk-term}([[\text{scCfFunction}\ T_2]\ T_3\ /\ L_3;\ T_4]) \end{array}}{\begin{array}{l} \text{table-per-term}(L,\ T_1,\ [[\text{scCfFunction}\ T_2]\ T_3\ /\ L_3;\ T_4]) = \\ \text{table-per-parameter}(L,\ T_1\ /\ L_3,\ T_5,\ T_6,\ T_4,\ 1) \end{array}} \qquad [\text{Tpt2}]$$

$$\text{table-per-term}(L,\ T_1,\ T_2)\ =\ \text{nil} \quad \textbf{otherwise} \qquad [\text{Tpt}]$$

Equation [*Tpp*1] will be applied as long as there is more than one parameter in the list of parameters, and the counter will be increased by one.

$$\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4;\ T_4{}',\ Int) = \qquad [\text{Tpp1}]$$
$$\begin{array}{l} \text{merge}(\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4,\ Int), \\ \qquad \text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4{}',\ Int + 1)) \end{array}$$

Equation [*Tpp*2] is applied if the parameter is annotated T_4/L_4. If the parameter is defined by a parameter on the left-hand side of the equation this parameter has been annotated during the restruction of the equation. T_4 is the parameter attribute representation of the parameter on the left-hand side of the equation. In this case we make a parameter attribute representation of the term declaration T_2 combined with the counter Int by means of the function cc (concatenate) and the result will be, for instance, "`par(tcs(SERIES,TENV)->BOOL,1)`"; T_5. Of the variable that represents the parameter, a term representation is made by means of function mk-term, for instance "`Tenv`";T_6. The variable T_1 is the constructor that was annotated in equation [*Tpt*2], T_3 is the term represention of the term. These objects are now put in a new entry of the table.

$$\frac{\begin{array}{l} T_5 = \text{par-att}(T_2,\ Int), \\ T_6 = \text{mk-term}(T_4) \end{array}}{\begin{array}{l} \text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4\ /\ L_4,\ Int) = \\ [[T_5\ T_1]\ L_4]\ /\ (L\ /\ T_3\ /\ T_6) \end{array}} \qquad [\text{Tpp2}]$$

Equation [*Tpp*3] applies if the parameter term is not annotated ($[T_4\ T_4{}']$). In this case it is not defined by a parameter term on the left-hand side of the equation but probably by a constant or a term. In this case, a string representation is made of the parameter term. T_6 represents both the definition of the parameter attribute as well as the string corresponding to the parameter attribute.

$$\frac{\begin{array}{l} T_5 = \text{par-att}(T_2,\ Int), \\ T_6 = \text{mk-term}([T_4\ T_4{}']) \end{array}}{\begin{array}{l} \text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ [T_4\ T_4{}'],\ Int) = \\ [[T_5\ T_1]\ T_6]\ /\ (L\ /\ T_3\ /\ T_6) \end{array}} \qquad [\text{Tpp3}]$$

$$\text{table-per-parameter}(L,\ T_1,\ T_2,\ T_3,\ T_4,\ Int)\ =\ \text{nil} \quad \textbf{otherwise} \qquad [\text{Tpp}]$$

E.5.2 WP

In the Module WP we check that the specification is well-presented.

module WP
imports ShortMessages$^{E.2.1}$
exports
  context-free syntax
    wp(ATerm, ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    wp5(ATerm, ATerm, ATerm) → ATerm
    wp6(ATerm, ATerm, ATerm) → ATerm
    compare-definitions(ATerm, ATerm) → ATerm
    last(ATerm) → ATerm
  variables
    Entry [*0-9*]∗ → ATerm
    Entries → ATerm
    Par [*0-9*]∗ → ATerm
    Def [*0-9*]∗ → ATerm
    Tab [*0-9*]∗ → ATerm
    Info [*0-9*]∗ → ATerm
    Def [*0-9*]∗ → ATerm
    TransDef [*0-9'*]∗ → ATerm
    ParConstr → ATerm
    Constr → ATerm
equations

Originally, all three arguments of the function wp are the same. They are all a copy of the table produced in the module WP-table and contain the definition rules for all parameter attributes in a specification. Every entry in the first table is compared with every entry in the second table by means of function wp5. The third copy of the table remains intact. The function wp5 is applied to check if property (v) in Definition 2.1 holds: two identical parameters defined for the same constructor must be defined by the same parameter term, constant or function.

$$\mathsf{wp}(\mathit{Entry}; \mathit{Entries}, \mathit{Tab}_1, \mathit{Tab}_2) = \mathsf{merge}(\mathsf{wp}(\mathit{Entry}, \mathit{Tab}_1, \mathit{Tab}_2), \mathsf{wp}(\mathit{Entries}, \mathit{Tab}_1, \mathit{Tab}_2)) \quad \text{[WP1]}$$

$$\mathsf{wp}(\mathit{Def}\ /\ \mathit{Info}, \mathit{Tab}_1, \mathit{Tab}_2) = \mathsf{wp5}(\mathit{Def}\ /\ \mathit{Info}, \mathit{Tab}_1, \mathit{Tab}_2) \quad \text{[WP2]}$$

The function wp5 defined below checks if two parameters with the same name and defined for the same child of a constructor, are defined equally. If the two parameters are not defined by the same term (equation [$WP53$]), the transitive definition will be found for both of these parameters by means of function wp6. These two transitive definition will be compared by means of function the function compare-definitions. The equation [$wp52$] is applied if two attributes are defined by the same term but occur in two different equations. We do not have to check parameters that are defined equally.

$$\mathsf{wp5}(\mathit{Entry}, \mathit{Entry}_1; \mathit{Entries}, \mathit{Tab}_2) = \mathsf{merge}(\mathsf{wp5}(\mathit{Entry}, \mathit{Entry}_1, \mathit{Tab}_2), \mathsf{wp5}(\mathit{Entry}, \mathit{Entries}, \mathit{Tab}_2)) \quad \text{[WP51]}$$

$$\mathsf{wp5}(\mathit{Def}\ /\ \mathit{Info}_1, \mathit{Def}\ /\ \mathit{Info}_2, \mathit{Tab}_2) = \mathsf{nil} \quad \text{[WP52]}$$

$$\frac{\begin{array}{c} \mathit{Par}_1 \neq \mathit{Par}_2, \\ \mathit{TransDef}_1 = \mathsf{wp6}(\mathit{Par}_1, \mathit{Tab}, \mathit{Tab}), \\ \mathit{TransDef}_2 = \mathsf{wp6}(\mathit{Par}_2, \mathit{Tab}, \mathit{Tab}), \\ \mathit{TransDef}_1 \neq \mathsf{nil}, \\ \mathit{TransDef}_2 \neq \mathsf{nil}, \\ \mathit{TransDef}_1{}' = \mathsf{merge}(\mathit{Info}_1, \mathit{TransDef}_1), \\ \mathit{TransDef}_2{}' = \mathsf{merge}(\mathit{Info}_2, \mathit{TransDef}_2) \end{array}}{\begin{array}{l} \mathsf{wp5}([\mathit{ParConstr}\ \mathit{Par}_1]\ /\ \mathit{Info}_1, [\mathit{ParConstr}\ \mathit{Par}_2]\ /\ \mathit{Info}_2, \mathit{Tab}) = \\ \mathsf{compare\text{-}definitions}(\mathit{TransDef}_1{}', \mathit{TransDef}_2{}') \end{array}} \quad \text{[WP53]}$$

$$\mathsf{wp5}(\mathit{Entry}_1, \mathit{Entry}_2, \mathit{Tab}) = \mathsf{nil} \quad \textbf{otherwise} \quad \text{[WP5]}$$

The function wp6 defined below searches for the transitive definition of a parameter attribute. When an attribute is found, the function wp6 is applied to the definition of that attribute. This goes on until an attribute is defined by a term or a constant, and the search is completed. This term or constant originally defines the attribute we started with and all the other attributes in the transitive definition for that matter. A transitive definition looks like: `"par(tcstat,1)"`;`"defined by"`; `"par(tcs,1)"`;`"defined by"`;`"tcd(Decls)"`.

$$\frac{\mathit{Par} \neq \mathit{Par}_1}{\begin{array}{l} \mathsf{wp6}(\mathit{Par}, [[\mathit{Par}\ \mathit{Constr}]\ \mathit{Par}_1]\ /\ \mathit{Info}; \mathit{Entries}, \mathit{Tab}) = \\ \mathsf{merge}(\mathit{Info}, \mathsf{wp6}(\mathit{Par}_1, \mathit{Tab}, \mathit{Tab})) \end{array}} \quad \text{[WP61]}$$

$$\frac{\mathit{Par} \neq \mathit{Par}_1}{\begin{array}{l} \mathsf{wp6}(\mathit{Par}, [[\mathit{Par}_1\ \mathit{Constr}]\ \mathit{Par}_2]\ /\ \mathit{Info}; \mathit{Entries}, \mathit{Tab}) = \\ \mathsf{wp6}(\mathit{Par}, \mathit{Entries}, \mathit{Tab}) \end{array}} \quad \text{[WP62]}$$

$$\begin{array}{l} \mathsf{wp6}(\mathit{Par}, [[\mathit{Par}\ \mathit{Constr}]\ \mathit{Par}]\ /\ \mathit{Info}; \mathit{Entries}, \mathit{Tab}) = \\ \mathsf{wp6}(\mathit{Par}, \mathit{Entries}, \mathit{Tab}) \end{array} \quad \text{[WP63]}$$

$$\frac{\mathit{Par} \neq \mathit{Par}_1}{\mathsf{wp6}(\mathit{Par}, [[\mathit{Par}\ \mathit{Constr}]\ \mathit{Par}_1]\ /\ \mathit{Info}, \mathit{Tab}) = \mathit{Info}} \quad \text{[WP64]}$$

$$\frac{\begin{array}{c} \mathit{Par} \neq \mathit{Par}_1, \\ \mathit{Par} \neq \mathit{Par}_2 \end{array}}{\mathsf{wp6}(\mathit{Par}, [[\mathit{Par}_1\ \mathit{Constr}]\ \mathit{Par}_2]\ /\ \mathit{Info}, \mathit{Tab}) = \mathsf{nil}} \quad \text{[WP65]}$$

$$\mathsf{wp6}(\mathit{Par}, [[\mathit{Par}\ \mathit{Constr}]\ \mathit{Par}]\ /\ \mathit{Info}, \mathit{Tab}) = \mathit{Info} \quad \text{[WP66]}$$

$$\mathsf{wp6}(\mathit{Par}, \mathit{Tab}_1, \mathit{Tab}_2) = \mathit{Par} \quad \textbf{otherwise} \quad \text{[WP6]}$$

The auxiliary function compare-definitions defined below removes all the transitive definitions that are equal or has the same constant or term as last object. If two transitive definitions are not removed they are returned and an error message can be created, containing the whole definition path to make the user understand what went wrong.

$$\mathsf{compare\text{-}definitions}(T, T) = \mathsf{nil} \quad \text{[CompD1]}$$

$$\mathsf{compare\text{-}definitions}(T, T') = \mathsf{nil} \quad \textbf{when} \quad \mathsf{last}(T) = \mathsf{last}(T') \quad \text{[CompD2]}$$

$$\mathsf{compare\text{-}definitions}(T, T') = [T\ T'] \quad \textbf{otherwise} \quad \text{[CompD]}$$

The function last defined below simply returns the last object in transitive definition.

$$\mathsf{last}(T; T_1) = \mathsf{last}(T_1) \quad \text{[Last1]}$$

$$\mathsf{last}(T) = T \quad \textbf{otherwise} \quad \text{[Last]}$$

Appendix F

The Storage Structure

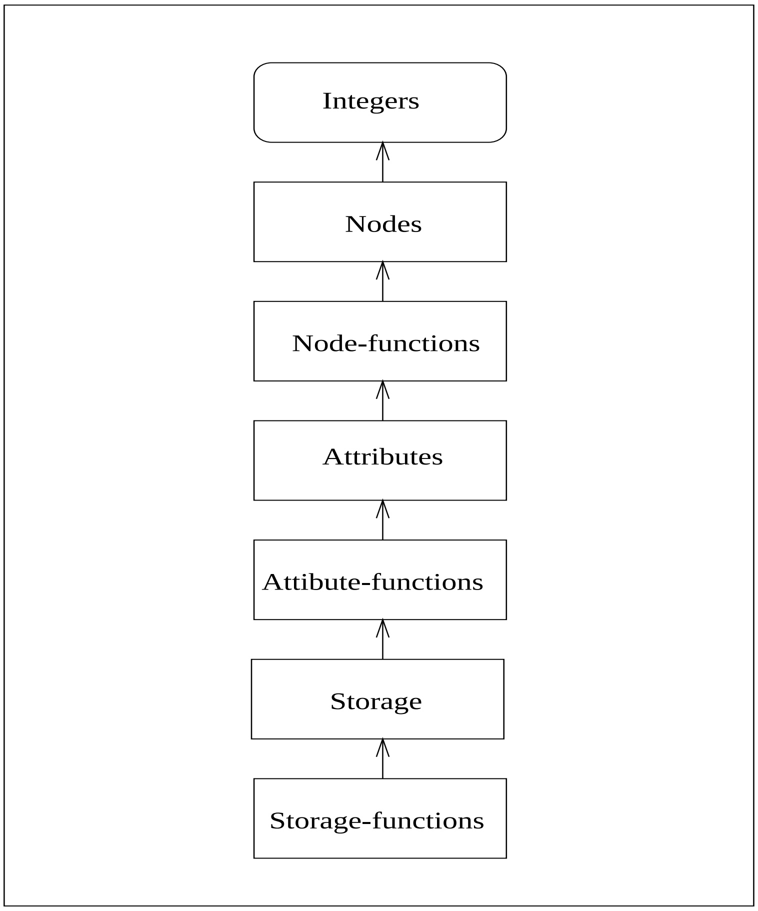


Figure F.1: Module import overview.

F.1 Nodes

In the module Nodes we have defined the nodes in the storage structure.

module **Nodes**
imports Integers$^{A.3}$

```
exports
  sorts NODE-PTR NODE-PTRS STATE NODE-CONTENT NODE
        NODES
  context-free syntax
    INT                        → NODE-PTR
    {INT ";"}*                 → NODE-PTRS

    init                       → STATE
    update                     → STATE
    non-inc-child-changed      → STATE
    top-new-subtree            → STATE
    idle                       → STATE

    cur-state STATE
    children {INT ";"}*        → NODE-CONTENT

    INT ":" NODE-CONTENT       → NODE
    {NODE ";"}*                → NODES
```

F.2 Node-functions

In the module Node-functions we have defined functions that can be applied to the nodes.

```
module Node-functions
imports Nodes[F.1]
exports
  context-free syntax
    make-node(INT, STATE, INT, {INT ";"}*)  → NODE
    state-of(NODE)                          → STATE
    children-of(NODE)                       → NODE-PTRS
    inc-children(NODE)                      → NODE-PTRS
    set-state(STATE, NODE)                  → NODE
    set-children({INT ";"}*, NODE)          → NODE
    new-children({INT ";"}*, INT, NODE)     → NODE
    make-children({INT ";"}*, INT)          → NODE-PTRS
    select-child(INT, NODE-PTR, NODES)      → NODE-PTR
    select-child-ptr(INT, NODE)             → NODE-PTR
    select-child-ptr(INT, {INT ";"}*)       → NODE-PTR
    find(INT, {NODE ";"}*)                  → NODES
    update(NODE, NODES)                     → NODES
    remove({INT ";"}*, NODES)               → NODES
    children-ptrs({INT ";"}*, NODES)        → NODE-PTRS
  variables
    NP [0-9']*    → INT
    NPs [0-9']*   → {INT ";"}*
    State [0-9']* → STATE
    NC [0-9']*    → NODE-CONTENT
    N [0-9']*     → NODE
    Ns [0-9']*    → {NODE ";"}*
    State [0-9']* → STATE
    R [0-9']*     → INT
    Rs [0-9']*    → {INT ";"}*
hiddens
  context-free syntax
```

filter({INT ";"}*) $\rightarrow$ NODE-PTRS

equations

Create a new node. First the node numbers of the children the node will have are determined by means of the function make-children. Pointers to these children are created before the actual nodes of these children are created. The state of the created node depends on the value of the variable *State*. The node number the created node is given depends on the variable *Int*, the pointer to this node. This pointer was determined when the node's parent was created. The variable *Rs* represents the ranks of the children the node will have.

$$\frac{\mathit{NPs} = \text{make-children}(\mathit{Rs}, \mathit{Int})}{\begin{array}{l}\text{make-node}(\text{NP}, \mathit{State}, \mathit{Int}, \mathit{Rs}) = \\ \text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}\end{array}} \quad \text{[MakeN]}$$

Below the projection functions defined for the nodes are given.

$$\text{state-of}(\text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}) = \mathit{State} \quad \text{[ProjN1]}$$

$$\text{children-of}(\text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}) = \mathit{NPs} \quad \text{[ProjN2]}$$

The function inc-children defined below returns the node-numbers of all children of a node that are incremental. Children with node number 0 are filtered.

$$\frac{\mathit{NPs} = \text{children-of}(N)}{\text{inc-children}(N) = \text{filter}(\mathit{NPs})} \quad \text{[IncCh]}$$

$$\text{filter}(\mathit{NPs}_1; 0; \mathit{NPs}_2) = \text{filter}(\mathit{NPs}_1; \mathit{NPs}_2) \quad \text{[Filter]}$$

$$\text{filter}(\mathit{NPs}) = \mathit{NPs} \quad \textbf{otherwise} \quad \text{[Filter]}$$

The functions set-state and set-children defined below can be used to give a node a new state or a new list of pointers to its children.

$$\begin{array}{l}\text{set-state}(\mathit{State}', \text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}) = \\ \text{NP} : \text{cur-state } \mathit{State}' \text{ children } \mathit{NPs}\end{array} \quad \text{[SetN1]}$$

$$\begin{array}{l}\text{set-children}(\mathit{NPs}', \text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}) = \\ \text{NP} : \text{cur-state } \mathit{State} \text{ children } \mathit{NPs}'\end{array} \quad \text{[SetN2]}$$

When the function new-children defined below is applied to a node, it is given a new list of pointers to new children. *Rs* are the ranks the children should have and *Int* is the last max in the storage. This max is used to detemine the node numbers of the new children.

$$\frac{\mathit{NPs} = \text{make-children}(\mathit{Rs}, \mathit{Int})}{\text{new-children}(\mathit{Rs}, \mathit{Int}, N) = \text{set-children}(\mathit{NPs}, N)} \quad \text{[NewC1]}$$

Below the function make-children is defined. Its first argument is a list of ranks and its second argument the previous max in the storage. For each rank in the list, a new node pointer is created. When a rank has value 0, the node has a non-incremental child for which no pointer will be created. This child will be referenced to by the nil pointer: 0.

$$\text{make-children}(, \mathit{Int}) = \quad \text{[MakeC1]}$$

$$\frac{\begin{array}{c}R \neq 0, \\ \mathit{Int}_1 = \mathit{Int} + 1, \\ \mathit{NPs} = \text{make-children}(\mathit{Rs}, \mathit{Int}_1)\end{array}}{\text{make-children}(R; \mathit{Rs}, \mathit{Int}) = \mathit{Int}_1; \mathit{NPs}} \quad \text{[MakeC2]}$$

$$\frac{\begin{array}{c}R = 0, \\ \mathit{NPs} = \text{make-children}(\mathit{Rs}, \mathit{Int})\end{array}}{\text{make-children}(R; \mathit{Rs}, \mathit{Int}) = R; \mathit{NPs}} \quad \text{[MakeC3]}$$

The function select-child defined below returns the node number of the *Int*'th child of node number *N* in the list of nodes, *Ns*.

$$\frac{\begin{array}{l} N;\ Ns_1 = \text{find}(\text{NP},\ Ns), \\ \text{NP}_1 = \text{select-child-ptr}(Int,\ N) \end{array}}{\text{select-child}(Int,\ \text{NP},\ Ns) = \text{NP}_1} \quad \text{[SelC1]}$$

The function select-child-ptr returns the node number of the *Int*'th child of the node *N*.

$$\frac{\begin{array}{l} Int \neq 0, \\ NPs = \text{children-of}(N), \\ \text{NP} = \text{select-child-ptr}(Int - 1,\ NPs) \end{array}}{\text{select-child-ptr}(Int,\ N) = \text{NP}} \quad \text{[SelCP1]}$$

$$\frac{Int = 0}{\text{select-child-ptr}(Int,\ \text{NP} : \text{NC}) = \text{NP}} \quad \text{[SelCP1]}$$

$$\frac{Int = 0}{\text{select-child-ptr}(Int,\ \text{NP};\ NPs) = \text{NP}} \quad \text{[SelCP3]}$$

$$\frac{Int \neq 0}{\begin{array}{l} \text{select-child-ptr}(Int,\ \text{NP};\ NPs) = \\ \text{select-child-ptr}(Int - 1,\ NPs) \end{array}} \quad \text{[SelCP4]}$$

Below the function find is defined. When this function is applied to a node number *N* and a list of nodes, the node with node number *N* is put as first object in the list and the list is returned.

$$\text{find}(\text{NP},\ Ns_1;\ \text{NP} : \text{NC};\ Ns_2) \ = \ \text{NP} : \text{NC};\ Ns_1;\ Ns_2 \quad \text{[FindN]}$$

In equation [UpdN] given below the function update is defined. The old node content NC_1 of the node with node number *NP* is replaced by the new node content *NC*.

$$\text{update}(\text{NP} : \text{NC},\ Ns_1;\ \text{NP} : \text{NC}_1;\ Ns_2) \ = \ \text{NP} : \text{NC}_1;\ Ns_1;\ Ns_2 \quad \text{[UpdN]}$$

The function remove defined below recursively removes all nodes referred to by the node pointer in its first argument in the list of nodes, its second argument.

$$\frac{N;\ Ns_1 = \text{find}(\text{NP},\ Ns)}{\begin{array}{l} \text{remove}(\text{NP};\ NPs,\ Ns) = \\ \text{remove}(NPs,\ Ns_1) \end{array}} \quad \text{[Remove]}$$

$$\text{remove}(NPs,\ Ns) \ = \ Ns \quad \textbf{otherwise} \quad \text{[Remove]}$$

Below the function children-ptrs is defined. This function returns a list of pointers to all node in the subtree which tops are of the nodes referred to by the node pointers in the first argument.

$$\frac{\begin{array}{r} N;\ Ns_1 = \text{find}(\text{NP},\ Ns), \\ NPs_1 = \text{inc-children}(N), \\ NPs_2 = \text{children-ptrs}(NPs,\ Ns_1), \\ NPs_3 = \text{children-ptrs}(NPs_1,\ Ns_1) \end{array}}{\text{children-ptrs}(\text{NP};\ NPs,\ Ns) = \text{NP};\ NPs_2;\ NPs_3} \quad \text{[CPtrs]}$$

$$\text{children-ptrs}(NPs,\ Ns) \ = \quad \textbf{otherwise} \quad \text{[CPtrs]}$$

F.3 Attributes

In the module Attributes we have defined the attributes we will use to store normal forms during incremental reduction.

module Attributes
imports Node-functions$^{F.2}$
exports

```
sorts ATT-NAME ATT-NAMES NORMAL-FORM ATT-TYPE STATUS
      ATT-PTR ATT-PTRS ATT-CONTENT ATTRIBUTE
      ATTRIBUTES
lexical syntax
  "\""~[\n\"]*"\"" → ATT-NAME
context-free syntax
  {ATT-NAME ";"}+             → ATT-NAMES
  "|" empty "|"               → NORMAL-FORM

  fun                         → ATT-TYPE
  par                         → ATT-TYPE

  "IN"                        → STATUS
  "TE"                        → STATUS
  "C"                         → STATUS
  "UC"                        → STATUS
  safe                        → STATUS
  unsafe                      → STATUS

  INT ":" {ATT-NAME ";"}+     → ATT-PTR
  {ATT-PTR ";"}*              → ATT-PTRS

  type ATT-TYPE
  status STATUS
  nf NORMAL-FORM
  preds {ATT-PTR ";"}*
  succs {ATT-PTR ";"}*
  corr-f-a {ATT-PTR ";"}*     → ATT-CONTENT
  ATT-PTR ":" ATT-CONTENT     → ATTRIBUTE
  {ATTRIBUTE ";"}*            → ATTRIBUTES
```

F.4 Attribute-functions

In the module Attribute-functions we have defined functions for the attributes.

module Attribute-functions
imports Attributes$^{F.3}$
exports

```
context-free syntax
  make-att(ATT-TYPE, {ATT-NAME ";"}+, NODE-PTR) → ATTRIBUTE

  type-of(ATTRIBUTE)                            → ATT-TYPE
  status-of(ATTRIBUTE)                          → STATUS
  nf-of(ATTRIBUTE)                              → NORMAL-FORM
  preds-of(ATTRIBUTE)                           → ATT-PTRS
  succs-of(ATTRIBUTE)                           → ATT-PTRS
  corr-f-atts-of(ATTRIBUTE)                     → ATT-PTRS
```

set-status(STATUS, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
set-nf(NORMAL-FORM, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
add-pred(ATT-PTR, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
add-succ(ATT-PTR, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
add-corr(ATT-PTR, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
rm-pred(ATT-PTR, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
rm-succ(ATT-PTR, ATTRIBUTE)	$\rightarrow$ ATTRIBUTE
fun-atts(INT, ATTRIBUTES)	$\rightarrow$ ATT-PTRS
par-atts(INT, ATTRIBUTES)	$\rightarrow$ ATT-PTRS
find(ATT-PTR, ATTRIBUTES)	$\rightarrow$ ATTRIBUTES
remove({INT ";"}*, ATTRIBUTES)	$\rightarrow$ ATTRIBUTES
rm-deps({INT ";"}*, ATT-CONTENT)	$\rightarrow$ ATT-CONTENT
safe-att(ATTRIBUTE)	$\rightarrow$ BOOL

variables

AN [*0-9'*]*	$\rightarrow$ ATT-NAME
ANs [*0-9'*]*	$\rightarrow$ {ATT-NAME ";"}*
Nf [*0-9'*]*	$\rightarrow$ NORMAL-FORM
Atype [*0-9'*]*	$\rightarrow$ ATT-TYPE
Status [*0-9'*]*	$\rightarrow$ STATUS
AP [*0-9'*]*	$\rightarrow$ ATT-PTR
APs [*0-9'*]*	$\rightarrow$ {ATT-PTR ";"}*
AC [*0-9'*]*	$\rightarrow$ ATT-CONTENT
A [*0-9'*]*	$\rightarrow$ ATTRIBUTE
As [*0-9'*]*	$\rightarrow$ {ATTRIBUTE ";"}*

equations

Create a new attribute of type *Atype* (function attribute: fun or parameter attribute: par), with name *ANs*. The created attribute is "attached" to node number *NP*. The combination of node number and attribute name is the reference to the created attribute.

make-att(*Atype*, *ANs*, NP) = [MakeN]
 NP : *ANs* : type *Atype* status IN nf | empty |
 preds succs corr-f-a

Below the definitions of the projection functions declared for the attributes are given.

type-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = *Atype* [ProjA1]

status-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = *Status* [ProjA2]

nf-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = *Nf* [ProjA3]

preds-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = APs_1 [ProjA4]

succs-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = APs_2 [ProjA5]

corr-f-atts-of(NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = APs_3 [ProjA6]

A part of an attribute can be given a new value when one of the set functions defined below is applied.

set-status(*Status'*, NP : *ANs* : type *Atype* status *Status* nf *Nf*
 preds APs_1 succs APs_2 corr-f-a APs_3) = [SetA1]

$$\begin{array}{l}\mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}'\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3\\ \mathsf{set\text{-}nf}(\mathit{Nf}', \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3) = \\ \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}'\\ \qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3\end{array} \tag{SetA2}$$

When the add functions defined below are called, a pointer to a predecessor, a successor or a corresponding function attribute of an attribute is added.

$$\begin{array}{l}\mathsf{add\text{-}pred}(\mathsf{AP}, \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3) = \\ \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathsf{AP}; \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3\end{array} \tag{AddP}$$

$$\begin{array}{l}\mathsf{add\text{-}succ}(\mathsf{AP}, \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3) = \\ \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathsf{AP}; \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3\end{array} \tag{AddS}$$

$$\begin{array}{l}\mathsf{add\text{-}corr}(\mathsf{AP}, \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_3) = \\ \mathsf{NP} : \mathit{ANs} : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2\ \mathsf{corr\text{-}f\text{-}a}\ \mathsf{AP}; \mathit{APs}_3\end{array} \tag{AddC}$$

When the functions rm-pred and rm-succ defined below are applied, the reference *AP* in the list of predecessors or successors in the attribute is removed.

$$\begin{array}{l}\mathsf{rm\text{-}pred}(\mathsf{AP}, \mathsf{AP}_1 : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1; \mathsf{AP}; \mathit{APs}_2\ \mathsf{succs}\ \mathit{APs}_3\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_4) = \\ \mathsf{AP}_1 : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathit{APs}_1; \mathit{APs}_2\ \mathsf{succs}\ \mathit{APs}_3\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_4\end{array} \tag{RMP}$$

$$\begin{array}{l}\mathsf{rm\text{-}succ}(\mathsf{AP}, \mathsf{AP}_1 : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad\qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2; \mathsf{AP}; \mathit{APs}_3\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_4) = \\ \mathsf{AP}_1 : \mathsf{type}\ \mathit{Atype}\ \mathsf{status}\ \mathit{Status}\ \mathsf{nf}\ \mathit{Nf}\\ \qquad \mathsf{preds}\ \mathit{APs}_1\ \mathsf{succs}\ \mathit{APs}_2; \mathit{APs}_3\ \mathsf{corr\text{-}f\text{-}a}\ \mathit{APs}_4\end{array} \tag{RMS}$$

When the function fun-atts is applied to a list of attributes and a node pointer *NP*, pointer to all function attributes in the list attached to the node with node number *NP* are returned.

$$\frac{\mathsf{fun} = \mathsf{type\text{-}of}(\mathsf{NP} : \mathit{ANs} : \mathsf{AC}),\quad \mathit{APs} = \mathsf{fun\text{-}atts}(\mathsf{NP}, \mathit{As}_1; \mathit{As}_2)}{\mathsf{fun\text{-}atts}(\mathsf{NP}, \mathit{As}_1; \mathsf{NP} : \mathit{ANs} : \mathsf{AC}; \mathit{As}_2) = \mathsf{NP} : \mathit{ANs}; \mathit{APs}} \tag{FunA1}$$

$$\mathsf{fun\text{-}atts}(\mathsf{NP}, \mathit{As}) = \quad \textbf{otherwise} \tag{FunA}$$

When the function par-atts is applied to a list of attributes and a node pointer *NP*, pointer to all parameter attributes in the list attached to the node with node number *NP* are returned.

$$\frac{\mathsf{par} = \mathsf{type\text{-}of}(\mathsf{NP} : \mathit{ANs} : \mathsf{AC}),\quad \mathit{APs} = \mathsf{par\text{-}atts}(\mathsf{NP}, \mathit{As}_1; \mathit{As}_2)}{\mathsf{par\text{-}atts}(\mathsf{NP}, \mathit{As}_1; \mathsf{NP} : \mathit{ANs} : \mathsf{AC}; \mathit{As}_2) = \mathsf{NP} : \mathit{ANs}; \mathit{APs}} \tag{FunA1}$$

$$\text{fun-atts}(\text{NP}, \mathit{As}) = \quad \textbf{otherwise} \qquad \text{[FunA]}$$

Below the definition of the function find is given. This function returns the list of attributes in which the attribute with attribute pointer AP is the head of this list.

$$\text{find}(\text{AP}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \text{AP} : \text{AC}; \mathit{As}_1; \mathit{As}_2 \qquad \text{[FindA]}$$

The function remove defined below removes all attributes attached to nodes which node numbers are in the list of node pointers, the first argument.

$$\text{remove}(\text{NP}; \mathit{NPs}, \mathit{As}_1; \text{NP} : \mathit{ANs} : \text{AC}; \mathit{As}_2) = \text{remove}(\text{NP}; \mathit{NPs}, \mathit{As}_1; \mathit{As}_2) \qquad \text{[Remove1]}$$

$$\text{remove}(, \mathit{As}) = \mathit{As} \qquad \text{[Remove2]}$$

$$\text{remove}(\text{NP}; \mathit{NPs}, \mathit{As}) = \text{remove}(\mathit{NPs}, \mathit{As}) \quad \textbf{otherwise} \qquad \text{[Remove]}$$

The function rm-deps defined below removes all depenencies in an attribute to attribute attached to nodes with its node number present in the list of node pointers, the first argument.

$$\begin{aligned}&\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}; \text{NP} : \mathit{ANs}; \mathit{APs}' \text{ succs } \mathit{APs}_2 \text{ corr-f-a } \mathit{APs}_3) = \qquad \text{[RMD1]}\\ &\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}; \mathit{APs}' \text{ succs } \mathit{APs}_2 \text{ corr-f-a } \mathit{APs}_3)\end{aligned}$$

$$\begin{aligned}&\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}_1 \text{ succs } \mathit{APs}; \text{NP} : \mathit{ANs}; \mathit{APs}' \text{ corr-f-a } \mathit{APs}_3) = \qquad \text{[RMD2]}\\ &\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}_1 \text{ succs } \mathit{APs}; \mathit{APs}' \text{ corr-f-a } \mathit{APs}_3)\end{aligned}$$

$$\begin{aligned}&\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}_1 \text{ succs } \mathit{APs}_2 \text{ corr-f-a } \mathit{APs}; \text{NP} : \mathit{ANs}; \mathit{APs}') = \qquad \text{[RMD3]}\\ &\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{type } \mathit{Atype} \text{ status } \mathit{Status} \text{ nf } \mathit{Nf}\\ &\qquad \text{preds } \mathit{APs}_1 \text{ succs } \mathit{APs}_2 \text{ corr-f-a } \mathit{APs}; \mathit{APs}')\end{aligned}$$

$$\text{rm-deps}(, \text{AC}) = \text{AC} \qquad \text{[RMD4]}$$

$$\text{rm-deps}(\text{NP}; \mathit{NPs}, \text{AC}) = \text{rm-deps}(\mathit{NPs}, \text{AC}) \quad \textbf{otherwise} \qquad \text{[RMD]}$$

The function safe-att defined below returns true when the input attribute has status safe (UC or C) and false when its status is unsafe (IN or TE).

$$\text{status-of}(A) = \text{UC} \Rightarrow \text{safe-att}(A) = \text{true} \qquad \text{[Stat1]}$$

$$\text{status-of}(A) = \text{C} \Rightarrow \text{safe-att}(A) = \text{true} \qquad \text{[Stat2]}$$

$$\text{status-of}(A) = \text{IN} \Rightarrow \text{safe-att}(A) = \text{false} \qquad \text{[Stat3]}$$

$$\text{status-of}(A) = \text{TE} \Rightarrow \text{safe-att}(A) = \text{false} \qquad \text{[Stat4]}$$

F.5 Storage

In the module Storage we have defined the storage structure we use when terms are applied to functions in an incremental specification.

module Storage
imports Attribute-functions$^{F.4}$

```
exports
  sorts STORAGE S-DIRECTIONS
  context-free syntax
    s-top NODE-PTR
    cur NODE-PTR
    max INT
    stack {INT ";"}*
    level INT
    path {INT ";"}*
    ch-atts {ATT-PTR ";"}*             → S-DIRECTIONS
    S-DIRECTIONS ":" {NODE ";"}*
    ":" {ATTRIBUTE ";"}*               → STORAGE
```

F.6 Storage-functions

In the module Storage-functions functions defined for the storage structure have been specified.

```
module Storage-functions
imports Storage^F.5
exports
  sorts BOOL-STORAGE NF-STORAGE RANKS INCINFO
  context-free syntax
    INT ":" INT                              → INCINFO
    BOOL ":" STORAGE                         → BOOL-STORAGE
    NORMAL-FORM ":" STORAGE                  → NF-STORAGE
    {INT ";"}*                               → RANKS

    make-storage()                           → STORAGE

    top-of(S-DIRECTIONS)                     → NODE-PTR
    cur-of(S-DIRECTIONS)                     → INT
    max-of(S-DIRECTIONS)                     → INT
    level-of(S-DIRECTIONS)                   → INT
    path-of(S-DIRECTIONS)                    → RANKS
    ch-atts-of(S-DIRECTIONS)                 → ATT-PTRS

    top-of-stack(S-DIRECTIONS)               → INT
    pop-of-stack(S-DIRECTIONS)               → S-DIRECTIONS
    push-cur-on-stack(S-DIRECTIONS)          → S-DIRECTIONS

    set-cur(NODE-PTR, S-DIRECTIONS)          → S-DIRECTIONS
    set-path({INT ";"}*, S-DIRECTIONS)       → S-DIRECTIONS
    set-stack({INT ";"}*, S-DIRECTIONS)      → S-DIRECTIONS
    set-level(INT, S-DIRECTIONS)             → S-DIRECTIONS
    set-new-max({INT ";"}*, S-DIRECTIONS)    → S-DIRECTIONS
    add-changed-att(ATT-PTR, S-DIRECTIONS)   → S-DIRECTIONS
    reset-ch-atts(S-DIRECTIONS)              → S-DIRECTIONS
    remove-first-path-rank(S-DIRECTIONS)     → S-DIRECTIONS

    remove-nodes-subtree(STORAGE)            → STORAGE

    increase({INT ";"}*, INT)                → INT
  variables
    S [0-9']*      → STORAGE
    SD [0-9']*     → S-DIRECTIONS
```

R $[0\text{-}9']*$ → INT
Rs $[0\text{-}9']*$ → {INT ";"}*
L $[0\text{-}9']*$ → INT
$IncInfo$ $[0\text{-}9']*$ → INCINFO

hiddens

context-free syntax

rm-deps(INT, {INT ";"}*, {ATTRIBUTES ";"}*) → ATTRIBUTES

equations

Create a new storage.

make-storage ( ) = [MakeS]
s-top 1 cur 0 max 1 stack level 0 path ch-atts : :

The definitions of projection functions for the storage directions are given below.

top-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = NP_1 [ProjS1]

cur-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = NP_2 [ProjS2]

max-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = Int_1 [ProjS3]

level-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = Int_2 [ProjS4]

path-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = Rs [ProjS5]

ch-atts-of(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = APs_1 [ProjS6]

The functions defined for the stack in the storage directions:

top-of-stack(s-top NP_1 cur NP_2 max Int_1 stack NP; NPs_1 level Int_2 path Rs ch-atts APs_1) = NP [Stack1]

pop-of-stack(s-top NP_1 cur NP_2 max Int_1 stack NP; NPs_1 level Int_2 path Rs ch-atts APs_1) = [Stack2]
s-top NP_1 cur NP_2 max Int_1 stack NPs_1
level $Int_2 - 1$ path Rs ch-atts APs_1

push-cur-on-stack(s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = [Stack3]
s-top NP_1 cur NP_2 max Int_1 stack NP_2; NPs_1
level $Int_2 + 1$ path Rs ch-atts APs_1

The set functions defined below can be used to give part of the storage directions a new value.

set-cur(NP′, s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = [SetSD1]
s-top NP_1 cur NP′ max Int_1 stack NPs_1
level Int_2 path Rs ch-atts APs_1

set-path(Rs', s-top NP_1 cur NP_2 max Int_1 stack NPs_1 level Int_2 path Rs ch-atts APs_1) = [SetSD2]
s-top NP_1 cur NP_2 max Int_1 stack NPs_1
level Int_2 path Rs' ch-atts APs_1

$$\begin{array}{l}\mathsf{set\text{-}new\text{-}max}(\mathit{Rs}', \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathsf{increase}(\mathit{Rs}', \mathit{Int}_1)\ \mathsf{stack}\ \mathit{NPs}_1 \\ \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1\end{array} \quad [\mathsf{SetSD3}]$$

$$\begin{array}{l}\mathsf{set\text{-}stack}(, \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack} \\ \mathsf{level}\ 0\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1\end{array} \quad [\mathsf{SetSD4}]$$

$$\begin{array}{l}\mathsf{set\text{-}level}(\mathit{Int}, \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \mathsf{level}\ \mathit{Int}\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1\end{array} \quad [\mathsf{SetSD5}]$$

The definition of functions defined for the list of changed attributes in the storage directions are given below. The function add-changed-att adds an attribute pointer to the list and the function reset-ch-atts resets the list; removes all pointers in the list.

$$\begin{array}{l}\mathsf{add\text{-}changed\text{-}att}(\mathsf{AP}, \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1;\ \mathsf{AP}\end{array} \quad [\mathsf{AddChA}]$$

$$\begin{array}{l}\mathsf{reset\text{-}ch\text{-}atts}(\mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\end{array} \quad [\mathsf{ResChA}]$$

The function remove-first-path-rank removes the first rank in the path in the storage directions.

$$\begin{array}{l}\mathsf{remove\text{-}first\text{-}path\text{-}rank}(\mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \qquad\qquad \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{R};\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1) = \\ \mathsf{s\text{-}top}\ \mathsf{NP}_1\ \mathsf{cur}\ \mathsf{NP}_2\ \mathsf{max}\ \mathit{Int}_1\ \mathsf{stack}\ \mathit{NPs}_1 \\ \mathsf{level}\ \mathit{Int}_2\ \mathsf{path}\ \mathit{Rs}\ \mathsf{ch\text{-}atts}\ \mathit{APs}_1\end{array} \quad [\mathsf{RFPR}]$$

The function remove-nodes-subtree defined below removes all subtrees of the current node in the storage. Dependencies between the attributes attached to the current node and attributes attached to its children are also removed.

$$\frac{\begin{array}{r@{\ }l}\mathsf{NP} &= \mathsf{cur\text{-}of}(\mathsf{SD}),\\ \mathit{N};\ \mathit{Ns}_1 &= \mathsf{find}(\mathsf{NP}, \mathit{Ns}),\\ \mathit{NPs} &= \mathsf{inc\text{-}children}(\mathit{N}),\\ \mathit{NPs}_1 &= \mathsf{children\text{-}ptrs}(\mathit{NPs}, \mathit{Ns}_1),\\ \mathit{Ns}_2 &= \mathsf{remove}(\mathit{NPs}_1, \mathit{Ns}_1),\\ \mathit{As}_1 &= \mathsf{remove}(\mathit{NPs}_1, \mathit{As}),\\ \mathit{As}_2 &= \mathsf{rm\text{-}deps}(\mathsf{NP}, \mathit{NPs}_1, \mathit{As}_1)\end{array}}{\begin{array}{l}\mathsf{remove\text{-}nodes\text{-}subtree}(\mathsf{SD} : \mathit{Ns} : \mathit{As}) = \\ \mathsf{SD} : \mathit{N};\ \mathit{Ns}_2 : \mathit{As}_2\end{array}} \quad [\mathsf{RMNS}]$$

The function rm-deps defined below removes dependencies to and from nodes with node number in the functions second argument in attributes attatched to the node with node number *NP*.

$$\frac{\begin{array}{r@{\ }l}\mathsf{AC}_1 &= \mathsf{rm\text{-}deps}(\mathit{NPs}, \mathsf{AC}),\\ \mathit{As}_3 &= \mathsf{rm\text{-}deps}(\mathsf{NP}, \mathit{NPs}, \mathit{As}_1;\ \mathit{As}_2)\end{array}}{\mathsf{rm\text{-}deps}(\mathsf{NP}, \mathit{NPs}, \mathit{As}_1;\ \mathsf{NP} : \mathit{ANs} : \mathsf{AC};\ \mathit{As}_2) = \mathsf{NP} : \mathit{ANs} : \mathsf{AC}_1;\ \mathit{As}_3} \quad [\mathsf{RMDeps}]$$

$$\text{rm-deps}(\mathsf{NP},\ \mathit{NPs},\ \mathit{As}) \ = \ \mathit{As} \quad \textbf{otherwise} \qquad \text{[RMDeps]}$$

The function increase defined below is intend to be used to increase the max in the storage. The first argument is a list of ranks of children. The max is increased by one for each rank in this list that is not equal to 0.

$$R \neq 0 \Rightarrow \quad \text{increase}(R;\ \mathit{Rs},\ \mathit{Int}) \ = \ \text{increase}(\mathit{Rs},\ \mathit{Int} + 1) \qquad \text{[Incr1]}$$

$$R = 0 \Rightarrow \quad \text{increase}(R;\ \mathit{Rs},\ \mathit{Int}) \ = \ \text{increase}(\mathit{Rs},\ \mathit{Int}) \qquad \text{[Incr2]}$$

$$\text{increase}(,\ \mathit{Int}) \ = \ \mathit{Int} \qquad \text{[Incr3]}$$

Appendix G

The module generations

G.1 Auxiliary transformation modules

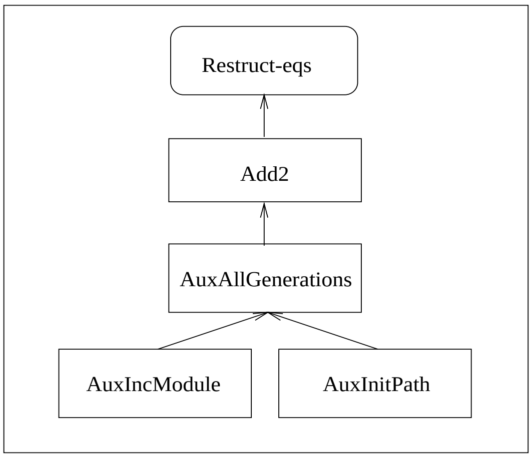


Figure G.1: Module import overview.

G.1.1 Add2

In the module Add2 functions are defined to add a new part of a section to a section in a module in AsFix-notation. If the section does not yet exist, a new section is created. A function for adding an argument to a function declaration is also defined in this module. The functions defined in this module are used when a module is transformed into an incremental module and when the InitUpdate module and the PathSearch module are derived.

module Add2
imports Restruct-eqs$^{D.6}$
exports
  context-free syntax
    add2imports(ATerm, ATerm) → ATerm
    add2hiddens(ATerm, ATerm, ATerm) → ATerm
    add2cfelems(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    add2cfsyntax(ATerm, ATerm) → ATerm
    add2variables(ATerm, ATerm) → ATerm

equations

The function add2imports defined below adds its second argument to the first import section in the syntax section (the first argument). If no import section is present in the syntax section, one is created.

$$\mathsf{add2imports}([\mathsf{scImports}\ T];\ T_1,\ T_2) = \mathsf{merge}([\mathsf{scImports}\ \mathsf{merge}(T_2,\ T)],\ T_1) \quad [\mathsf{A2Im1}]$$

$$\mathsf{add2imports}([\mathsf{scImports}\ T],\ T_1) = [\mathsf{scImports}\ \mathsf{merge}(T_1,\ T)] \quad [\mathsf{A2Im2}]$$

$$\mathsf{add2imports}([L\ T];\ T_1,\ T_2) = \mathsf{merge}([L\ T],\ \mathsf{add2imports}(T_1,\ T_2)) \quad [\mathsf{A2Im3}]$$

when

$L \neq \mathsf{scImports}$

$$\mathsf{add2imports}([L\ T],\ T_1) = \mathsf{merge}([L\ T],\ [\mathsf{scImports}\ T_1]) \quad [\mathsf{A2Im4}]$$

when

$L \neq \mathsf{scImports}$

The function add2hiddens defined below adds a new part of a context-free syntax section (T_1) and new variables (T_2) in the hidden part of a syntax section in a module by means of the functions add2cfsyntax and add2variables. If no hidden section exists in the syntax section one is created, equation $[A2H4]$.

$$\mathsf{add2hiddens}([\mathsf{scHiddens}\ T];\ T',\ T_1,\ T_2) = \mathsf{merge}(\mathsf{add2hiddens}([\mathsf{scHiddens}\ T],\ T_1,\ T_2),\ T') \quad [\mathsf{A2H1}]$$

$$\mathsf{add2hiddens}([L\ T];\ T',\ T_1,\ T_2) = \mathsf{merge}([L\ T],\ \mathsf{add2hiddens}(T',\ T_1,\ T_2)) \quad [\mathsf{A2H2}]$$

when

$L \neq \mathsf{scHiddens}$

$$\frac{T_3 = \mathsf{add2cfsyntax}(T,\ T_1),\quad T_4 = \mathsf{add2variables}(T_3,\ T_2)}{\mathsf{add2hiddens}([\mathsf{scHiddens}\ T],\ T_1,\ T_2) = [\mathsf{scHiddens}\ T_4]} \quad [\mathsf{A2H3}]$$

$$\frac{L \neq \mathsf{scHiddens},\quad T_3 = [\mathsf{scContextFreeSyntax}\ T_1],\quad T_4 = [\mathsf{scVariables}\ T_2]}{\mathsf{add2hiddens}([L\ T],\ T_1,\ T_2) = \mathsf{merge}([L\ T],\ [\mathsf{scHiddens}\ \mathsf{merge}(T_3,\ T_4)])} \quad [\mathsf{A2H4}]$$

The function add2cfelems defined below adds the sort represented by T_1 to the context free elements in the declaration of a function, represented by the input ATerm T.

$$\mathsf{add2cfelems}([\mathsf{scCfElems}\ T],\ T_1) = [\mathsf{scCfElems}\ \mathsf{add2cfelems}(T,\ T_1)] \quad [\mathsf{A2CfE1}]$$

$$\mathsf{add2cfelems}(T_1;\ T_2,\ T_3) = T_1;\ \mathsf{add2cfelems}(T_2,\ T_3) \quad [\mathsf{A2CfE2}]$$

$$\mathsf{add2cfelems}(\texttt{")"},\ T_3) = \texttt{","};\ \mathsf{merge}(T_3,\ \texttt{")"}) \quad [\mathsf{A2CfE3}]$$

$$\mathsf{add2cfelems}(T,\ T_3) = T;\ T_3 \qquad \textbf{otherwise} \quad [\mathsf{A2CfE}]$$

When the function add2cfsyntax defined below is applied, its second argument is added to the first context-free syntax section in its first argument. If no context-free syntax section is present, one is created.

$$\mathsf{add2cfsyntax}([\mathsf{scContextFreeSyntax}\ T];\ T_1,\ T_2) = \mathsf{merge}([\mathsf{scContextFreeSyntax}\ \mathsf{merge}(T_2,\ T)],\ T_1) \quad [\mathsf{A2CfS1}]$$

$$\mathsf{add2cfsyntax}([\mathsf{scContextFreeSyntax}\ T],\ T_1) = [\mathsf{scContextFreeSyntax}\ \mathsf{merge}(T_1,\ T)] \quad [\mathsf{A2CfS2}]$$

$$\mathsf{add2cfsyntax}([L\ T];\ T_1,\ T_2) = \mathsf{merge}([L\ T],\ \mathsf{add2cfsyntax}(T_1,\ T_2)) \quad [\mathsf{A2CfS3}]$$

when

$L \neq \mathsf{scContextFreeSyntax}$

$$\text{add2cfsyntax}([L\ T],\ T_1) \qquad \text{[A2CfS4]}$$
$$= \text{merge}([L\ T],\ [\text{scContextFreeSyntax}\ T_1])$$
$$\textbf{when}$$
$$L \neq \text{scContextFreeSyntax}$$

When the function add2variables defined below is applied, its second argument is added to the first variable declaration section in its first argument. If no such section is present, one is created.

$$\text{add2variables}([\text{scVariables}\ T];\ T_1,\ T_2) = \text{merge}([\text{scVariables merge}(T_2,\ T)],\ T_1) \qquad \text{[A2Var1]}$$
$$\text{add2variables}([\text{scVariables}\ T],\ T_1) = [\text{scVariables merge}(T_1,\ T)] \qquad \text{[A2Var2]}$$
$$\text{add2variables}([L\ T];\ T_1,\ T_2) = \text{merge}([L\ T],\ \text{add2variables}(T_1,\ T_2)) \qquad \text{[A2Var3]}$$
$$\textbf{when}\quad L \neq \text{scVariables}$$
$$\text{add2variables}([L\ T],\ T_1) = \text{merge}([L\ T],\ [\text{scVariables}\ T_1]) \qquad \text{[A2Var4]}$$
$$\textbf{when}\quad L \neq \text{scVariables}$$

G.1.2 AuxAllGenerations

In the module AuxAllGeneration functions used in all the generations of modules have been defined.

module AuxAllGenerations
imports Add2$^{G.1.1}$
exports
context-free syntax

constructor-sorts(ATerm)	→ ATerm
rmdoubles(ATerm)	→ ATerm
mk-variables(ATerm)	→ ATerm
mk-numbered-variables(ATerm)	→ ATerm
sort-of(ATerm)	→ ATerm
first-sort(ATerm)	→ ATerm
new-var(ATerm, ATerm)	→ ATerm
sort2var(ATerm)	→ ATerm
sort2var(ATerm, ATerm)	→ ATerm
next-var-index(ATerm)	→ ATerm
lex-term(ATerm, ATerm)	→ ATerm
var-term(ATerm, ATerm)	→ ATerm
var-term(ATerm, ATerm, ATerm)	→ ATerm
tag(ATerm, INT)	→ ATerm
bool-constant(ATerm)	→ ATerm
sctrue	→ Literal
scfalse	→ Literal
scBOOL	→ Literal
scATT-NAME	→ Literal
scSTORAGE	→ Literal
scBOOL-STORAGE	→ Literal
scINT	→ Literal

equations

The function constructor-sorts defined below returns the the sorts of all constructors in the input function declarations. The constructor is the first argument in a function thus the sort of this first argument is selected.

constructor-sorts(T_1; T_2) = [CSS1]
merge(constructor-sorts(T_1), constructor-sorts(T_2))

constructor-sorts([scCfFunction T_1; T_2; T_3]) = [scSort first-sort(T_1)] [CSS2]
constructor-sorts(T) = nil **otherwise** [CSS]

Below the function rm-doubles is defined. When this function has been applied to a list of objects, all objects but one have been removed if these objects were the same object.

rmdoubles(T_1; T_2) = rmdoubles(T_2) **when** occurs(T_1, T_2) = true [RD1]
rmdoubles(T_1; T_2) = T_1; rmdoubles(T_2) **when** occurs(T_1, T_2) = false [RD2]
rmdoubles(T) = T **otherwise** [RD]

The function mk-variables defined below returns for each sort in the input ATerm T a new definition of a variable of this sort.

mk-variables(T_1; T_2) = merge(mk-variables(T_1), mk-variables(T_2)) [Vs1]

mk-variables([scSort T]) = [Vs2]
[scVariable [scLexElems [scSort cc(T; "var")]]; [scSort T]]

mk-variables(T) = nil **otherwise** [Vs]

The function mk-numbered-variables defined below returns for each sort in the input ATerm T a new definition of a variable of this sort such that an index can be given to the variable.

mk-numbered-variables(T_1; T_2) = [NVs1]
merge(mk-numbered-variables(T_1), mk-numbered-variables(T_2))
mk-numbered-variables([scSort T]) = [NVs2]
[scVariable
 [scLexElems
  [scSort cc(T; "var")];
  [scLexIter
   [scCharClass "[0-9]"]; "*"]];
 [scSort T]]

mk-numbered-variables(L) = mk-numbered-variables([scSort L]) [NVs3]
mk-numbered-variables(T) = nil **otherwise** [NVs]

The function sort-of defined below yields the Literal representing the sort of the input ATerm. The input expected is an ATerm representing an ASF+SDF-term.

sort-of([scCfFunction T_1; [L T_2]; T_3]) = T_2 [SO1]
sort-of([scIter [L_1 T_1]; T_2; L_2]) = T_1 [SO2]
sort-of([scVar T_1; [L T_2]]) = T_2 [SO3]
sort-of([scLex T_1; [L T_2]]) = T_2 [SO4]
sort-of([scSort T]) = T [SO5]
sort-of(L) = nil [SO6]
sort-of(nil) = nil [SO7]
sort-of([T_1 T_2]) = sort-of(T_1) **otherwise** [SO]

The function first-sort defined below returns the literal representing the first sort in an ATerm. This function should be applied to ATerms representing the context-free elements of a function.

$$\begin{aligned}
&\mathsf{first\text{-}sort}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]) &=&\ \mathsf{first\text{-}sort}(T_1) && \text{[FS1]}\\
&\mathsf{first\text{-}sort}([\mathsf{scCfElems}\ T]) &=&\ \mathsf{first\text{-}sort}(T) && \text{[FS2]}\\
&\mathsf{first\text{-}sort}(L;\ T) &=&\ \mathsf{first\text{-}sort}(T) && \text{[FS3]}\\
&\mathsf{first\text{-}sort}([\mathsf{scSort}\ L];\ T) &=&\ L && \text{[FS4]}\\
&\mathsf{first\text{-}sort}([\mathsf{scSort}\ L]) &=&\ L && \text{[FS5]}
\end{aligned}$$

The function new-var defined below can be used when variables are generated of, for instance, names of sorts. The created variables have the form $\mathit{SORTvar}_x$ where SORT is the first argument and x is an index. The input expected by this function is the name of the to create variable and a list of earlier created variables. When a new variable is created, the list of "old" varibles is check. If the variable is present in this list it will be given an index increased by one with respect to the variable in the list with the same name.

$$\frac{T_1 = \mathsf{sort2var}(T,\ \texttt{"1"})}{\mathsf{new\text{-}var}(T,\ \mathsf{nil}) = T_1\ /\ (T\ /\ \texttt{"1"};\ \mathsf{nil})} \qquad \text{[NewV1]}$$

$$\frac{\begin{array}{c} L' = \mathsf{next\text{-}var\text{-}index}(L),\\ T_1 = \mathsf{sort2var}(T,\ L') \end{array}}{\mathsf{new\text{-}var}(T,\ T\ /\ L;\ T_2) = T_1\ /\ (T\ /\ L';\ T_2)} \qquad \text{[NewV2]}$$

$$\frac{\begin{array}{c} T \neq T',\\ T_1\ /\ T_2{}' = \mathsf{new\text{-}var}(T,\ T_2) \end{array}}{\mathsf{new\text{-}var}(T,\ T'\ /\ L;\ T_2) = T_1\ /\ (T'\ /\ L;\ T_2{}')} \qquad \text{[NewV3]}$$

The function next-var-index increases a Literal representing an integer by converting it to an integer, increasing the integer by one and then convert it back into a Literal.

$$\mathsf{next\text{-}var\text{-}index}(T) \ = \ \mathsf{int2lit}(\mathsf{lit2int}(T) + 1) \qquad \text{[NVI]}$$

The function sort2var defined below returns a variable representation of the input. If the input is SORT, the ATerm representing the variable *SORTvar* is returned. This function can be invoked with one or two arguments. When it is invoked with two arguments, the second argument is an ATerm representing the index the variable should have and is given.

$$\begin{aligned}
&\mathsf{sort2var}(L) &=&\ [\mathsf{scVar}\ \mathsf{cc}(L;\ \texttt{"var"});\ [\mathsf{scSort}\ L]] && \text{[S2Var1]}\\
&\mathsf{sort2var}([\mathsf{scSort}\ L]) &=&\ \mathsf{sort2var}(L) && \text{[S2Var2]}\\
&\mathsf{sort2var}(L_1,\ L_2) &=&\ [\mathsf{scVar}\ \mathsf{cc}(L_1;\ \texttt{"var"};\ L_2);\ [\mathsf{scSort}\ L_1]] && \text{[S2Var3]}\\
&\mathsf{sort2var}([\mathsf{scSort}\ L_1],\ L_2) &=&\ \mathsf{sort2var}(L_1,\ L_2) && \text{[S2Var4]}
\end{aligned}$$

The function *lex-term* defined by equation [LexT] given below yields an ATerm representing a lexical L_1 of sort L_2.

$$\begin{aligned}
&\mathsf{lex\text{-}term}(L_1,\ L_2) = && \text{[LexT]}\\
&\ [\mathsf{scLex}\ L_1;\ [\mathsf{scSort}\ L_2]]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{var\text{-}term}(T_1,\ T_2) = && \text{[Var1]}\\
&\ [\mathsf{scVar}\ T_1;\ [\mathsf{scSort}\ T_2]]\\
&\mathsf{var\text{-}term}(T_1,\ T_2,\ T_3) = && \text{[Var2]}\\
&\ [\mathsf{scVar}\ \mathsf{cc}(T_1;\ T_2);\ [\mathsf{scSort}\ T_3]]
\end{aligned}$$

tag(L, Int) = cc(L; int2lit(Int)) [TAG]

The function bool-constant yields an ATerm representing the ASF+SDF constant true of false depending on the input.

```
bool-constant(T) =                                        [MT]
 [[scCfFunction
   [scCfElems T];
   [scSort scBOOL];
   [scAttributes nil]]
  nil]
```

The following equations turns the introduced constants into strings.

aterms(sctrue)	=	"true"	[AT1]
aterms(scfalse)	=	"false"	[AT2]
aterms(scBOOL)	=	"BOOL"	[AT3]
aterms(scATT-NAME)	=	"ATT-NAME"	[AT4]
aterms(scSTORAGE)	=	"STORAGE"	[AT5]
aterms(scBOOL-STORAGE)	=	"BOOL-STORAGE"	[AT6]
aterms(scINT)	=	"INT"	[AT7]

G.1.3 AuxIncModule

In the module AuxIncModule global function used during the generation of incremental modules have been defined. These functions are used when modules are created for both the external and the ASF+SDF version.

module AuxIncModule
imports AuxAllGenerations$^{G.1.2}$
exports
  context-free syntax
    mk-new-normalforms(ATerm) → ATerm
    imp2incimp(ATerm, ATerm) → ATerm
    conversion-sorts(ATerm) → ATerm
    rank-table(ATerm) → ATerm
    add-ranks-level(ATerm, ATerm) → ATerm
    sort2nf(ATerm) → ATerm
    rm-inc-att(ATerm) → ATerm
    rank-level-R-L() → ATerm
hiddens
  context-free syntax
    conv-sorts(ATerm) → ATerm
    remove(ATerm, ATerm) → ATerm
    table(ATerm, INT) → ATerm
    get-rank(ATerm, ATerm) → ATerm
    rank-and-level(ATerm) → ATerm
    incinfo-decl() → ATerm
exports
  context-free syntax
    scNORMAL-FORM → Literal

scSafe	$\rightarrow$ Literal
scUnSafe	$\rightarrow$ Literal
scstore-fun-att	$\rightarrow$ Literal
scstore-par-att	$\rightarrow$ Literal
scINCINFO	$\rightarrow$ Literal
scretrieve	$\rightarrow$ Literal

equations

The function mk-new-normalforms defined below returns for each sort in the input ATerm T one function declarations: A function declaration for converting the sort to a normal form.

mk-new-normalforms(T_1; T_2) =
merge(mk-new-normalforms(T_1), mk-new-normalforms(T_2)) [MNnf1]

mk-new-normalforms([scSort T]) = sort2nf(T) [MNnf2]

mk-new-normalforms(T) = nil **otherwise** [MNnf]

The function imp2incimp defined below transforms the imports section in a module, meaning that it adds the prefix Inc to the names of the imported modules that have also been transformed or will be transformed. The names of these modules should be listed in the input ATerm T_2.

imp2incimp(T; T_1, T_2) = merge(imp2incimp(T, T_2), imp2incimp(T_1, T_2)) [Imp1]
imp2incimp([scId T], T_2) = [scId cc("Inc"; T)] [Imp2]
when
occurs(T, T_2) = true

imp2incimp(T, T_2) = T **otherwise** [Imp]

The function conversion-sorts defined below selects all sorts for which conversion functions must be declared. These sorts are the result sorts of functions that have been declared as incremental and the sorts of their parameters. Because it is not simple to select only the sorts of the parameters in the context-free elements in a function declaration in AsFix notation, all sorts in the incremental function are selected by means of the function conv-sorts and then the sorts of the constructors in the functions are selected, thus the sort of the first argument. Given these to list of sorts, the "constructor sorts" are removed in the list of all sorts by means of function remove.

T_1 = conv-sorts(T),
T_2 = constructor-sorts(T)
———————————————
conversion-sorts(T) =
rmdoubles(remove(T_2, T_1)) [CvS1]

The function rank-table defined by equation $[Rt1] - [Rt3]$ expects as input an ATerm representing the left-hand side of an equation. The variables representing the constuctor are annotated with an index starting from the left by means of the function table. A list with these annotated "variables" is returned.

rank-table([[scCfFunction T] [T_1 T_2]; T_3]) =
rank-table([[scCfFunction T] [T_1 T_2]]) [Rt1]

rank-table([[scCfFunction T] [T_1 T_2; T_3]]) = table(T_2; T_3, 1) [Rt2]
rank-table([[scCfFunction T] [T_1 [L T_2]]]) = [L T_2] / int2lit(0) [Rt3]

The function add-rank-level defined below adds the rank and the level to all (sub) terms having an incremental top operator in an equation. The rank is the child number of the constructor in a term and the level added is simly *L+1*. The second argument is a table with ATerms representing variables that have been annotated with the rank indicating which child they represent. Equation $[ARL1]-[ARL4]$ are used to isolate the terms in an equation. The equation $[ARL5]$ applies when the term has an incremental top operator and has parameters. It is possible that these parameters (T_5) contain other terms with incremental top operators thus the function add-ranks-level is applied to these parameters. Equation $[ARL6]$ applies when a term has a non-incremental top operator but has parameters. As in equation $[ARL5]$, the function add-ranks-level must be applied to these parameters terms. The equation $[ARL7]$ applies when a term has an incremental top operator but no parameters.

$$\text{add-ranks-level}(T_1;\ T_2,\ T) \ = \ \text{merge}(\text{add-ranks-level}(T_1,\ T),\ \text{add-ranks-level}(T_2,\ T)) \qquad \text{[ARL1]}$$

$$\begin{array}{l}\text{add-ranks-level}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ T) = \\ \ [\text{scCondEquation } L_1;\ L_2;\ T_1;\ \text{add-ranks-level}(T_2,\ T);\ \text{add-ranks-level}(T_3,\ T)]\end{array} \qquad \text{[ARL2]}$$

$$\text{add-ranks-level}([\text{scConditions } T_1],\ T) \ = \ [\text{scConditions add-ranks-level}(T_1,\ T)] \qquad \text{[ARL3]}$$

$$\begin{array}{l}\text{add-ranks-level}([\text{scCondition } L;\ T_1;\ T_2],\ T) = \\ \ [\text{scCondition } L;\ \text{add-ranks-level}(T_1,\ T);\ \text{add-ranks-level}(T_2,\ T)]\end{array} \qquad \text{[ARL4]}$$

$$\frac{\begin{array}{r}\text{true} = \text{isInc}(T_3), \\ T_1{}' = \text{add2cfelems}(T_1,\ [\text{scSort scINCINFO}]), \\ T_6 = \text{rank-and-level}(\text{get-rank}(T_4,\ T)), \\ T_5{}' = \text{add-ranks-level}(T_5,\ T), \\ T_6{}' = \text{merge}(T_5{}',\ T_6)\end{array}}{\begin{array}{l}\text{add-ranks-level}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4;\ T_5],\ T) = \\ \ [[\text{scCfFunction } T_1{}';\ T_2;\ T_3]\ T_4;\ T_6{}']\end{array}} \qquad \text{[ARL5]}$$

$$\frac{\begin{array}{r}\text{false} = \text{isInc}(T_3), \\ T_4{}' = \text{add-ranks-level}(T_4;\ T_5,\ T)\end{array}}{\begin{array}{l}\text{add-ranks-level}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4;\ T_5],\ T) = \\ \ [[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4{}']\end{array}} \qquad \text{[ARL6]}$$

$$\frac{\begin{array}{r}\text{true} = \text{isInc}(T_3), \\ T_1{}' = \text{add2cfelems}(T_1,\ [\text{scSort scINCINFO}]), \\ T_6 = \text{rank-and-level}(\text{get-rank}([T_4\ \ T_5],\ T)), \\ T_4{}' = \text{merge}([T_4\ \ T_5],\ T_6)\end{array}}{\begin{array}{l}\text{add-ranks-level}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ [T_4\ \ T_5]],\ T) = \\ \ [[\text{scCfFunction } T_1{}';\ T_2;\ T_3]\ T_4{}']\end{array}} \qquad \text{[ARL7]}$$

$$\text{add-ranks-level}(T_1,\ T) \ = \ T_1 \quad \textbf{otherwise} \qquad \text{[ARL]}$$

The function sort2nf defined below yields an ATerm representing a declaration of a conversion function: |SORT| →NORMAL-FORM.

$$\begin{array}{l}\text{sort2nf}(T) = \\ \ [\text{scCfFunction} \\ \ \ [\text{scCfElems} \\ \ \ \ [\text{scQLit "|"}];\ [\text{scSort } T];\ [\text{scQLit "|"}]]; \\ \ \ [\text{scSort scNORMAL-FORM}]; \\ \ \ [\text{scAttributes nil}]]\end{array} \qquad \text{[S2N]}$$

The function conv-sorts defined below yields a list of ATerms representing all the sorts that occured in the input.

$$\begin{array}{llll}
\text{conv-sorts}(T_1;\ T_2) & = \text{merge}(\text{conv-sorts}(T_1), \text{conv-sorts}(T_2)) & & [\text{CS1}] \\
\text{conv-sorts}([\text{scSort}\ T]) & = [\text{scSort}\ T] & & [\text{CS2}] \\
\text{conv-sorts}([T_1\ T_2]) & = \text{merge}(\text{conv-sorts}(T_1), \text{conv-sorts}(T_2)) & \textbf{when}\ T_1 \neq \text{scSort} & [\text{CS3}] \\
\text{conv-sorts}(T) & = \text{nil} & \textbf{otherwise} & [\text{CS}]
\end{array}$$

The function remove defined below returns a list of all objects in its second arguments that do not occur in its first argument.

$$\text{remove}(T,\ T_1;\ T_2) = \text{merge}(\text{remove}(T,\ T_1), \text{remove}(T,\ T_2)) \quad [\text{RM1}]$$

$$\begin{array}{llll}
\text{remove}(T, [T_1\ T_2]) & = [T_1\ T_2] & \textbf{when}\ \text{false} = \text{occurs}([T_1\ T_2], T) & [\text{RM2}] \\
\text{remove}(T,\ T_1) & = \text{nil} & \textbf{otherwise} & [\text{RM}]
\end{array}$$

Below the function table is defined. The objects in the first argument are annotated with an index and a list of these annotated objects is returned.

$$\begin{array}{llll}
\text{table}(T_1;\ T_2,\ Int) & = T_1\ /\ \text{int2lit}(Int); \text{table}(T_2,\ Int + 1) & & [\text{Tab1}] \\
\text{table}(T,\ Int) & = T\ /\ \text{int2lit}(Int) & \textbf{otherwise} & [\text{Tab}]
\end{array}$$

The function get-rank defined below is an auxilary function that searches a table (or a list) with ATerms representing variables that have been annotated with a rank. Its first argument is an ATerm representing a variable. When this variable is found in the table, the annotation, the rank, is returned.

$$\begin{array}{llll}
\text{get-rank}(T,\ T\ /\ T_1;\ T_2) & = T_1 & & [\text{GR1}] \\
\text{get-rank}(T,\ T\ /\ T_1) & = T_1 & & [\text{GR2}] \\
\text{get-rank}(T,\ T'\ /\ T_1;\ T_2) & = \text{get-rank}(T,\ T_2) & \textbf{when}\ T \neq T' & [\text{GR1}] \\
\text{get-rank}(T,\ T_1) & = \texttt{"WHATNOW"} & \textbf{otherwise} & [\text{GR}]
\end{array}$$

The function rank-and-level defined below returns an AsFix term representing the Asf+Sdf term *R*:*L+1* where *R* is the rank represented by *T*.

$$\begin{array}{ll}
\text{rank-and-level}(T) = & [\text{RaL}] \\
\ [\text{incinfo-decl}\ (\ ) & \\
\ \ \text{lex-term}(T, \text{scINT}); & \\
\ \ [[\text{scCfFunction} & \\
\ \ \ \ [\text{scCfElems} & \\
\ \ \ \ \ [\text{scSort scINT}]; [\text{scQLit}\ \texttt{"+"}]; [\text{scSort scINT}]]; & \\
\ \ \ \ [\text{scSort scINT}]; [\text{scAttributes nil}]] & \\
\ \ \ \text{var-term}(\texttt{"L"}, \text{scINT}); \text{lex-term}(\texttt{"1"}, \text{scINT})]] &
\end{array}$$

The function rm-inc-att defined below expects as input a list of function attributes and removes the attribute incremental out of this list. If the list only contains the attribute incremental, the empty list nil is returned.

$$\begin{array}{llll}
\text{rm-inc-att}(\text{scincremental};\ T_1) & = T_1 & & [\text{RIA1}] \\
\text{rm-inc-att}(T;\ T_1) & = \text{merge}(T, \text{rm-inc-att}(T_1)) & \textbf{when}\ T \neq \text{scincremental} & [\text{RIA2}] \\
\text{rm-inc-att}(\text{scincremental}) & = \text{nil} & & [\text{RIA3}] \\
\text{rm-inc-att}(L) & = L & \textbf{when}\ L \neq \text{scincremental} & [\text{RIA4}] \\
\text{rm-inc-att}(\text{nil}) & = \text{nil} & & [\text{RIA5}]
\end{array}$$

The function rank-level-R-L yields an ATerm representing the ASF+SDF term *R*:*L*.

rank-level-R-L ( ) = [MII]
[incinfo-decl ( ) var-term("R", scINT); var-term("L", scINT)]

The function incinfo-decl returns an ATerm representing the declaration: INT ":" INT → INC-INFO.

incinfo-decl ( ) = [IIDecl]
[scCfFunction
[scCfElems
[scSort scINT]; ":"; [scSort scINT]];
[scSort scINCINFO];
[scAttributes nil]]

The following equations turns the introduced constants into strings.

aterms(scNORMAL-FORM)	=	"NORMAL-FORM"	[AT1]
aterms(scSafe)	=	"status-is-safe"	[AT2]
aterms(scUnSafe)	=	"status-is-unsafe"	[AT3]
aterms(scstore-fun-att)	=	"store-fun-att"	[AT4]
aterms(scstore-par-att)	=	"store-par-att"	[AT5]
aterms(scINCINFO)	=	"INCINFO"	[AT6]
aterms(scretrieve)	=	"retrieve"	[AT7]

G.1.4 AuxInitPath

In the module AuxInitPath some functions have been defined that are used during the generation of both the InitUpdate module as well as the PathSearch module.

module AuxInitPath
imports AuxAllGenerations$^{G.1.2}$
exports
context-free syntax

iter-decl(ATerm, ATerm, ATerm) → ATerm
varR() → ATerm
varR(ATerm) → ATerm
injections(ATerm) → ATerm

equations

The function iter-decl returns an ATerm representing the declaration of an iteration.

iter-decl(L_1, L_2, L_3) = [ID]
[scIter [scSort L_1]; [scQLit L_2]; L_3]

varR ( ) = var-term("R", scINT) [VarR1]
varR(L) = var-term("R", L, scINT) [VarR2]

The function injections returns all injections that can be detected in the left-hand side of an equation given a list of equations. Output: SORT1/SORT2 when the injection SORT1 → SORT2 has been detected.

$$\mathsf{injections}(T_1;\ T_2) = \mathsf{merge}(\mathsf{injections}(T_1),\ \mathsf{injections}(T_2)) \quad \text{[INJ1]}$$

$$\mathsf{injections}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \mathsf{injections}(\mathsf{constructor}(T_1)) \quad \text{[INJ2]}$$

$$\frac{\mathsf{true} = \mathsf{isInjection}([[\mathsf{scCfFunction}\ \ T_1;\ T_2;\ T_3]\ \ T_4])}{\begin{array}{l}\mathsf{injections}([[\mathsf{scCfFunction}\ \ T_1;\ T_2;\ T_3]\ \ T_4]) = \\ {[\mathsf{scSort}\ \mathsf{first\text{-}sort}(T_1)]}\ /\ T_2\end{array}} \quad \text{[INJ3]}$$

$$\mathsf{injections}(T)\ =\ \mathsf{nil}\quad \textbf{otherwise} \quad \text{[INJ]}$$

G.2 Transform module, external version

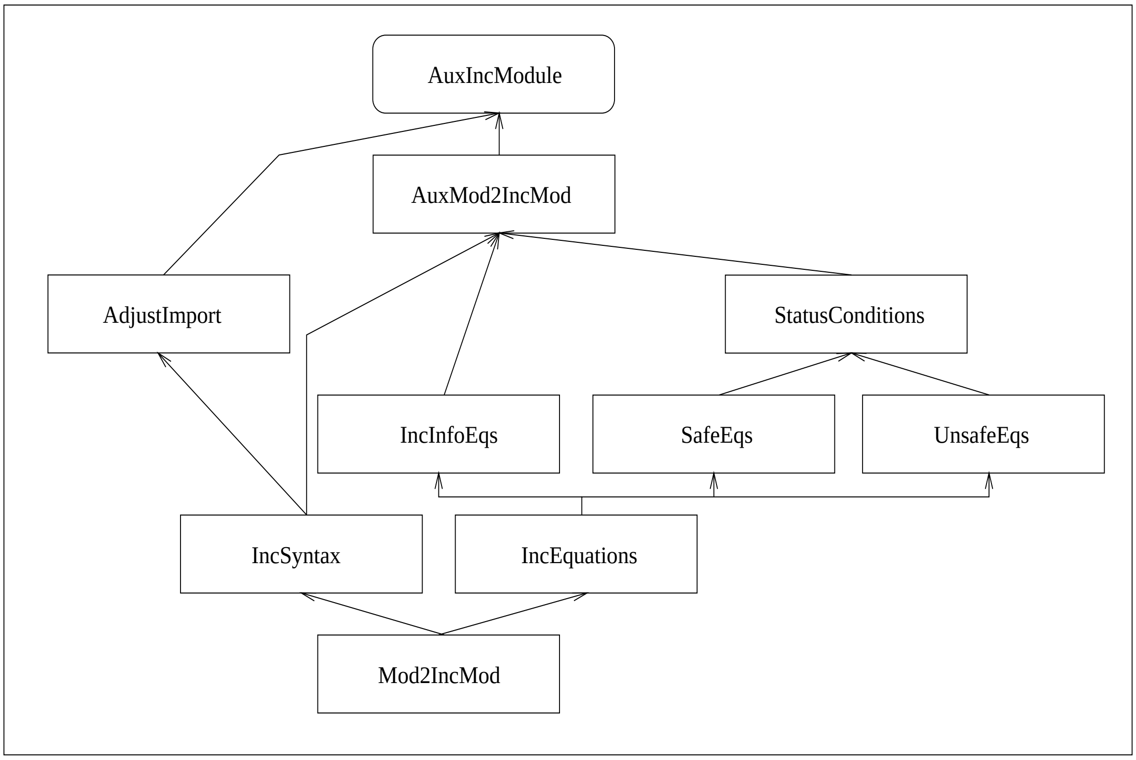


Figure G.2: Module import overview.

G.2.1 Mod2IncMod

In the module Mod2IncMod the transformation of a module into an incremental module is specified. The result of this transformation can be used when the storage for incremental reduction is implemented in a conventional programming language for which the implementation of the rewrite system supports external functions.

module **Mod2IncMod**
imports IncSyntax$^{G.2.2}$ IncEquations$^{G.2.4}$
exports
  context-free syntax
    mod2incmod(ATerm, ATerm) $\rightarrow$ ATerm
    mod2incmod(ATerm) $\rightarrow$ ATerm
equations

The function mod2incmod defined below transforms a module into an incremental module. This means that functions that are defined as incremental and the equations defined for these functions are transformed in such a way that when terms are applied to the transformed equations, they can be reduced incrementally. The input ATerm represents a module in AsFix notation and the input ATerm T_5 contains a list of the names of the other modules that are transformed. We need this information to be able to change the names of imported modules that are transformed or are going to be transformed. In the first condition, all declarations of functions declared as incremental in the module are selected by means of the function inc-operators. The sorts for which conversion functions are needed (sorts that will be stored and retrieved during reduction) are selected by means of function conversion-sorts. At this point we are also interested in the sorts of the constructors, the first arguments in incremental functions. These sorts are selected when the function constructor-sorts is applied. In the fourth condition the name of the module is changed by adding the prefix "Inc". The source syntax and the source equations are used to restore comment. When a module in AsFix-representation has been changed, the comments can no longer be restored and these two parts are given the new value nil. The syntax section and the equations in the module are transformed by means of the functions syn2incsyn and eqs2inceqs respectively. The equation [MIE2] applies when no list of names of other transformed is supplied.

$$\frac{\begin{array}{l} T_6 = \text{inc-operators}(\text{cf-syntax}(T_2)), \\ T_7 = \text{conversion-sorts}(T_6), \\ T_8 = \text{constructor-sorts}(T_6), \\ L_1{}' = \text{cc}(\texttt{"Inc"};\ L_1), \\ T_1{}' = [\text{scSourceSyntax nil}], \\ T_2{}' = \text{syn2incsyn}(T_2,\ T_5,\ T_7,\ T_8), \\ T_3{}' = [\text{scSourceEquations nil}], \\ T_4{}' = \text{eqs2inceqs}(T_4,\ T_7,\ T_6) \end{array}}{\begin{array}{l} \text{mod2incmod}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4],\ T_5) = \\ [\text{scModule } L_1{}';\ L_2;\ T_1{}';\ T_2{}';\ T_3{}';\ T_4{}'] \end{array}} \quad \text{[MIE1]}$$

$$\text{mod2incmod}(T) \ = \ \text{mod2incmod}(T,\ \texttt{""}) \quad \text{[MIE2]}$$

G.2.2 IncSyntax

In the module IncSyntax the transformation of the syntax section of a module is given.

module IncSyntax

imports AdjustImport$^{G.2.3}$ AuxMod2IncMod$^{G.2.9}$

exports

  context-free syntax

    syn2incsyn(ATerm, ATerm, ATerm, ATerm) → ATerm

hiddens

  context-free syntax

    trans-fun-decls(ATerm) → ATerm
    mk-conversion-fun-decl(ATerm) → ATerm

equations

The function syn2incsyn defined below transforms the syntax section in a module into an incremental syntax section. The input ATerm T contains the syntax sections of a module in AsFix notation, T_1 contains the names of other modules that have been, or shall be, transformed. T_2 contains the sorts for which conversion functions and new NORMAL-FORMs must be declared. T_3 contain a list of the sorts of the constructors that occur in the module. First, the import section is adjusted by means of function adjust-import. This function yields a new copy of the syntax section. The declarations of the functions that have been declared as incremental are transformed when the function trans-fun-decls is applied. Declarations of the conversion functions are created (mk-conversion-fun-decl) and variables are defined for all the conversion sorts and the sorts of the constructors in the fourth condition by means of the the function mk-variables. The name of the module in which the functions for manipulating the storage are defined is added to the import section in the module by means of the function add2imports. Finally, the declarations of the conversion functions and the variables are added to a context-free syntax section and a variable section in a hidden part of the module. This is done when the function add2hiddens is applied. If no hidden section is present in the module, one will be created.

$$\frac{\begin{array}{l} T' = \mathsf{adjust\text{-}import}(T,\ T_1), \\ T'' = \mathsf{trans\text{-}fun\text{-}decls}(T'), \\ T_4 = \mathsf{mk\text{-}new\text{-}normalforms}(T_2), \\ T_5 = \mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T_2), \\ T_6 = \mathsf{mk\text{-}variables}(\mathsf{merge}(T_2,\ T_3)), \\ T_7 = \mathsf{add2imports}(T'',\ [\mathsf{scId}\ \texttt{"StorageInterfaceReduce"}]), \\ T_8 = \mathsf{add2hiddens}(T_7,\ \mathsf{merge}(T_4,\ T_5),\ T_6) \end{array}}{\begin{array}{l} \mathsf{syn2incsyn}([\mathsf{scPrefixSyntax}\ T],\ T_1,\ T_2,\ T_3) = \\ [\mathsf{scPrefixSyntax}\ T_8] \end{array}} \quad \text{[SIS1]}$$

The function trans-fun-decls defined below transforms the declarations of funtions that have been declared as incremental. In equation [TFD6] the attribute incremental is removed when the function rm-inc-att is applied to the attributes of the function. A copy is made of the function declaration and to this copy the sort INCINFO is added as a new argument. This function thus returns two function declarations, one in which the only adjustment was to remove the attribute incremental and the second to which the INCINFO has been added as an argument.

$$\begin{array}{l} \mathsf{trans\text{-}fun\text{-}decls}(T_1;\ T_2) = \\ \mathsf{merge}(\mathsf{trans\text{-}fun\text{-}decls}(T_1),\ \mathsf{trans\text{-}fun\text{-}decls}(T_2)) \end{array} \quad \text{[TFD1]}$$

$$\mathsf{trans\text{-}fun\text{-}decls}([\mathsf{scImports}\ T]) = [\mathsf{scImports}\ T] \quad \text{[TDF2]}$$

$$\mathsf{trans\text{-}fun\text{-}decls}([\mathsf{scExports}\ T]) = [\mathsf{scExports}\ \mathsf{trans\text{-}fun\text{-}decls}(T)] \quad \text{[TDF3]}$$

$$\mathsf{trans\text{-}fun\text{-}decls}([\mathsf{scHiddens}\ T]) = [\mathsf{scHiddens}\ \mathsf{trans\text{-}fun\text{-}decls}(T)] \quad \text{[TDF4]}$$

$$\begin{array}{l} \mathsf{trans\text{-}fun\text{-}decls}([\mathsf{scContextFreeSyntax}\ T]) = \\ [\mathsf{scContextFreeSyntax}\ \mathsf{trans\text{-}fun\text{-}decls}(T)] \end{array} \quad \text{[TFD5]}$$

$$\frac{\begin{array}{l} \mathsf{true} = \mathsf{isInc}(T_3), \\ T_1{}' = \mathsf{addINCINFO}(T_1), \\ T_3{}' = \mathsf{rm\text{-}inc\text{-}att}(T_3) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}fun\text{-}decls}([\mathsf{scCfFunction}\ T_1;\ T_2;\ [L\ T_3]]) = \\ [\mathsf{scCfFunction}\ T_1;\ T_2;\ [L\ T_3{}']];\ [\mathsf{scCfFunction}\ T_1{}';\ T_2;\ [L\ T_3{}']] \end{array}} \quad \text{[TFD6]}$$

$$\mathsf{trans\text{-}fun\text{-}decls}(T) = T \quad \textbf{otherwise} \quad \text{[TFD]}$$

The function mk-conversion-fun-decl defined below returns for each sort in the input ATerm T a function declarations: A declaration for a function to convert a normal form to the sort.

$$\begin{array}{l} \mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T_1;\ T_2) = \\ \mathsf{merge}(\mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T_1),\ \mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T_2)) \end{array} \quad \text{[CFD1]}$$

$$\mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}([\mathsf{scSort}\ T]) = \mathsf{nf2sort}(T) \quad \text{[CFD2]}$$

$$\mathsf{mk\text{-}conversion\text{-}fun\text{-}decl}(T) = \mathsf{nil} \quad \textbf{otherwise} \quad \text{[CFD]}$$

G.2.3 AdjustImport

In the module AdjustImport the changes that have to be made to the names of imported modules have been specified.

module AdjustImport
imports AuxIncModule$^{G.1.3}$
exports
  context-free syntax
    adjust-import(ATerm, ATerm) $\rightarrow$ ATerm
equations
The function adjust-import defined below changes the names of other transformed modules that are imported by the module that is being transformed by means of the function imp2incimp. T_1 contains the names of these modules.

$$\text{adjust-import}(T;\ T',\ T_1) = \text{merge}(\text{adjust-import}(T,\ T_1),\ \text{adjust-import}(T',\ T_1)) \quad \text{[AImp1]}$$

$$\text{adjust-import}([\text{scImports}\ T],\ T_1) = [\text{scImports imp2incimp}(T,\ T_1)] \quad \text{[AImp2]}$$

$$\text{adjust-import}([L\ T],\ T_1) = [L\ T] \quad \textbf{when}\ L \neq \text{scImports} \quad \text{[AImp3]}$$

$$\text{adjust-import}(T,\ T_1) = T \quad \textbf{otherwise} \quad \text{[AImp]}$$

G.2.4 IncEquations

In the module IncEqs the transformation of equations is specified. The result of this transformation are equations that can be used during incremental reduction.

module IncEquations
imports IncInfoEqs$^{G.2.5}$ SafeEqs$^{G.2.6}$ UnsafeEqs$^{G.2.7}$
exports
  context-free syntax
    eqs2inceqs(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    trans-eqs(ATerm) $\rightarrow$ ATerm
    mk-conversion-eqs(ATerm, INT) $\rightarrow$ ATerm
equations
The function eqs2inceqs defined below transforms the equations in the equation section of a module. T is a list of equations, T_1 is a list of the declaration of the functions that were declared as incremental in the syntax section of the module and T_2 is a list of the sorts for which conversion function were declared and for which equations must be added. In the first condition *indicating* equations are created by means of the function inc-info-eqs. In the second condition equations are created for the case that the value of a related attribute is reliable. In the next condition the existing equations in the prefix syntax are transformed and finally equations are created to define the conversion functions. The results of the four conditions are merged together and form the new equation section.

$$\frac{\begin{array}{l} T_3 = \text{inc-info-eqs}(T_2,\ T,\ 1), \\ T_6 = \text{safeeqs}(T_2,\ T,\ 1), \\ T_4 = \text{trans-eqs}(T), \\ T_5 = \text{mk-conversion-eqs}(T_1,\ 1) \end{array}}{\begin{array}{l} \text{eqs2inceqs}([\text{scPrefixEquations}\ T],\ T_1,\ T_2) = \\ [\text{scPrefixEquations merge}(T_3,\ \text{merge}(T_6,\ \text{merge}(T_4,\ T_5)))] \end{array}} \quad \text{[E2IE]}$$

The function trans-eqs defined below transforms all equations in the module. Equations which top operator has not been declared as incremental are left intact. If the top operator in an equation has been declared as incremental an equation is created which first argument invokes a function status-is-unsafe during reduction. Before the unsafe equation can be created by means of the function unsafeeq, the rank and the level is added to the left-hand side of the equation and the incremental terms in the conditions and on the right-hand side of the equation by means of the function add-ranks-level.

$$\text{trans-eqs}(T_1;\ T_2) \ = \ \text{merge}(\text{trans-eqs}(T_1), \text{trans-eqs}(T_2)) \qquad \text{[TEq1]}$$

$$\frac{\text{true} = \text{isInc}(\text{operator}(T_1)),\quad T_4 = \text{unsafeeq}(\text{add-ranks-level}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3], \text{rank-table}(T_1)))}{\text{trans-eqs}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = T_4} \qquad \text{[TEq2]}$$

$$\text{trans-eqs}(T) \ = \ T \quad \textbf{otherwise} \qquad \text{[TEq]}$$

The function mk-conversion-eqs defined below returns for each sort in the input ATerm T an equation defining the conversion function normal form to the sort.

mk-conversion-eqs(T; T_1, Int) = [CEq1]
merge(mk-conversion-eqs(T, Int), mk-conversion-eqs(T_1, $Int + 1$))
mk-conversion-eqs([scSort T], Int) = [CEq2]
[scCondEquation
scImplies; tag("Cv", Int);
[nf2sort(T) [sort2nf(T) sort2var(T)]];
sort2var(T);
[scConditions nil]]

$$\text{mk-conversion-eqs}(T, Int) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[CEq]}$$

G.2.5 IncInfoEqs

In the module IncInfoEqs the generation of equations introducing the incremental information has been specified.

module IncInfEqs
imports AuxMod2IncMod$^{G.2.9}$
exports
context-free syntax
inc-info-eqs(ATerm, ATerm, INT) → ATerm
equations

Below the function inc-info-eqs is defined. This function has two arguments. The first argument is a list of all function declarations of the functions in the module that have been declared as incremental. The second argument is a list containing all the equations in the module. For each function declaration one defining equation is selected. The left-hand side of the selected equation is kept and the rest of the equation wasted. A copy is made of this left-hand side and the incremental information, the INCINFO, is added to this copy. This INCINFO is represented in the copy by *1:0*, thus the rank is 1 and the level is 0. The copy is made the right-hand side of the equation, thus when reduction starts, the rank and the level in the term is initialised with the values 1 and 0 respectively. The third argument in the function inc-info-eqs is used to number these equations.

$$\text{inc-info-eqs}(T_1;\ T_2,\ T_3,\ \mathit{Int}) = \quad \text{[IIEq1]}$$
$$\text{merge}(\text{inc-info-eqs}(T_1,\ T_3,\ \mathit{Int}),\ \text{inc-info-eqs}(T_2,\ T_3,\ \mathit{Int}+1))$$

$$\text{inc-info-eqs}(T,\ [\text{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) = \quad \text{[IIEq2]}$$
$$\text{inc-info-eqs}(T,\ [\text{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3],\ \mathit{Int})$$

$$\text{inc-info-eqs}(T,\ [\text{scCondEquation}\ L_1;\ L_2;\ [T_1\ T_1'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) \quad \text{[IIEq3]}$$
$$= \text{inc-info-eqs}(T,\ T_4,\ \mathit{Int})$$
when
$$T \neq T_1$$

$$\frac{\begin{array}{l} T_1' = \text{addINCINFO}(T_1), \\ T_7 = \text{sort2var}(\text{first-sort}(T_1)), \\ T_4' = \text{merge}(T_9,\ \text{lex-term}(\texttt{"1:0"},\ \text{scINCINFO})), \\ T_6' = [\text{scConditions nil}], \\ L_3' = \text{tag}(\texttt{"AII"},\ \mathit{Int}) \end{array}}{\begin{array}{l} \text{inc-info-eqs}([L\ T_1;\ T_2;\ T_3],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ T_3]\ T_8;\ T_9];\ T_5;\ T_6],\ \mathit{Int}) = \\ [L_1\ L_2;\ L_3';\ [[L\ T_1;\ T_2;\ T_3]\ T_7;\ T_9];\ [[L\ T_1';\ T_2;\ T_3]\ T_7;\ T_4'];\ T_6'] \end{array}} \quad \text{[IIEq4]}$$

$$\frac{\begin{array}{l} T_1' = \text{addINCINFO}(T_1), \\ T_7 = \text{sort2var}(\text{first-sort}(T_1)), \\ T_4' = \text{lex-term}(\texttt{"1:0"},\ \text{scINCINFO}), \\ T_6' = [\text{scConditions nil}], \\ L_3' = \text{tag}(\texttt{"AII"},\ \mathit{Int}) \end{array}}{\begin{array}{l} \text{inc-info-eqs}([L\ T_1;\ T_2;\ T_3],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ T_3]\ [T_8\ T_9]];\ T_5;\ T_6],\ \mathit{Int}) = \\ [L_1\ L_2;\ L_3';\ [[L\ T_1;\ T_2;\ T_3]\ T_7];\ [[L\ T_1';\ T_2;\ T_3]\ T_7;\ T_4'];\ T_6'] \end{array}} \quad \text{[IIEq4]}$$

$$\text{inc-info-eqs}(T,\ T_1,\ \mathit{Int})\ =\ \text{nil} \quad \textbf{otherwise} \quad \text{[IIEq]}$$

G.2.6 SafeEqs

In the module SafeEqs the function safeeeqs has been specified. When this function is applied, the equations for the safe case during incremental reduction are generated.

module SafeEqs
imports StatusCondition$^{G.2.8}$
exports
  context-free syntax
    safeeqs(ATerm, ATerm, INT) → ATerm
hiddens
  context-free syntax
    mk-new-rhs-safe(ATerm, ATerm) → ATerm
    mk-retrieve(ATerm) → ATerm
equations

The function safeeqs defined below is similar to the function inc-info-eqs and have the same arguments. For each function declaration one defining equation is selected. In the left-hand side of this equation the variables representing the constructor are replaced by a new variable and the INC-INFO is added as an argument. A condition is created by means of the function mk-status-cond.

In this condition a function will be invoked during reduction to check that the value of the related attribute is reliable or safe. A new right-hand side for the equation is created in which the function retrieve will be invoked during reduction of a term to retrieve the value from the storage. The function retrieve returns an item of sort NORMAL-FORM thus this normal form must be converted into the result sort of the function for which the equation is defined. These two actions are accomplished when the function mk-new-rhs-safe is applied. Equation [SEq4] applies when the term on left-hand side of the equation has parameters and when there are no parameters in this term, equation [SEq5] applies.

$$\mathsf{safeeqs}(T_1;\ T_2,\ T_3,\ Int)\ =\ \mathsf{merge}(\mathsf{safeeqs}(T_1,\ T_3,\ Int),\ \mathsf{safeeqs}(T_2,\ T_3,\ Int+1)) \qquad \text{[SEq1]}$$

$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3];\ T_4,\ Int) = \qquad \text{[SEq2]}$$
$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3],\ Int)$$

$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T_1\ T_1'];\ T_2;\ T_3];\ T_4,\ Int) \qquad \text{[SEq3]}$$
$$= \mathsf{safeeqs}(T,\ T_4,\ Int)$$
when
$$T \neq T_1$$

$$\frac{\begin{array}{l} L_2' = \mathsf{scImplies}, \\ L_3' = \mathsf{tag}(\texttt{"default-SIS"},\ Int), \\ T_1' = \mathsf{addINCINFO}(T_1), \\ T_3' = \mathsf{rm\text{-}inc\text{-}att}(T_3), \\ T_4' = \mathsf{sort2var}(\mathsf{first\text{-}sort}(T_1)), \\ T_5' = \mathsf{merge}(T_5,\ \mathsf{rank\text{-}level\text{-}R\text{-}L}\ (\ )), \\ T_8 = \mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([L\ T_1;\ T_2;\ T_3])), \\ T_6' = \mathsf{mk\text{-}new\text{-}rhs\text{-}safe}(T_8,\ \mathsf{sort\text{-}of}(T_2)), \\ T_7' = [\mathsf{scConditions}\ \mathsf{mk\text{-}status\text{-}cond}(T_8,\ \mathsf{scSafe})] \end{array}}{\begin{array}{l} \mathsf{safeeqs}([L\ T_1;\ T_2;\ [L_4\ T_3]],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ [L_4\ T_3]]\ T_4;\ T_5];\ T_6;\ T_7],\ Int) = \\ [L_1\ L_2';\ L_3';\ [[L\ T_1';\ T_2;\ [L_4\ T_3']]\ T_4';\ T_5'];\ T_6';\ T_7'] \end{array}} \qquad \text{[SEq4]}$$

$$\frac{\begin{array}{l} L_2' = \mathsf{scImplies}, \\ L_3' = \mathsf{tag}(\texttt{"default-SIS"},\ Int), \\ T_1' = \mathsf{addINCINFO}(T_1), \\ T_3' = \mathsf{rm\text{-}inc\text{-}att}(T_3), \\ T_4' = \mathsf{sort2var}(\mathsf{first\text{-}sort}(T_1)), \\ T_5' = \mathsf{rank\text{-}level\text{-}R\text{-}L}\ (\ ), \\ T_8 = \mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([L\ T_1;\ T_2;\ T_3])), \\ T_6' = \mathsf{mk\text{-}new\text{-}rhs\text{-}safe}(T_8,\ \mathsf{sort\text{-}of}(T_2)), \\ T_7' = [\mathsf{scConditions}\ \mathsf{mk\text{-}status\text{-}cond}(\mathsf{scSafe},\ T_8)] \end{array}}{\begin{array}{l} \mathsf{safeeqs}([L\ T_1;\ T_2;\ [L_4\ T_3]],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ [L_4\ T_3]]\ [T_4\ T_5]];\ T_6;\ T_7],\ Int) = \\ [L_1\ L_2';\ L_3';\ [[L\ T_1';\ T_2;\ [L_4\ T_3']]\ T_4';\ T_5'];\ T_6';\ T_7'] \end{array}} \qquad \text{[SEq5]}$$

$$\mathsf{safeeqs}(T,\ T_1,\ Int)\ =\ \mathsf{nil}\quad \textbf{otherwise} \qquad \text{[SEq]}$$

The function mk-new-rhs-safe defined below returns an ATerm representing the right-hand side of a *safe* incremental equation. T represents the function attribute and T_1 represents the result sort of the top operator on the left-hand side of the original equation.

$$\mathsf{mk\text{-}new\text{-}rhs\text{-}safe}(T,\ T_1) = \qquad \text{[NrhsS]}$$
$$[\mathsf{nf2sort}(T_1)\ \mathsf{mk\text{-}retrieve}(T)]$$

The function mk-retrieve yields an ATerm representing the ASF+SDF term retrieve(AN) where AN is the name of the attribute which value is to be retrieved.

```
mk-retrieve(T) =                                                    [MR]
 [[scCfFunction
   [scCfElems scretrieve; "("; [scSort scATT-NAME]; ")"];
   [scSort scNORMAL-FORM];
   [scAttributes nil]]
  lex-term(T, scATT-NAME)]
```

G.2.7 UnsafeEqs

In the module UnsafeEqs the specification is given for the generation of equations for the case when the status of an attribute is "unreliable".

module UnsafeEqs
imports StatusCondition$^{G.2.8}$
exports
  context-free syntax
    unsafeeq(ATerm) → ATerm
hiddens
  context-free syntax

```
    mk-new-lhs-eq(ATerm)                          → ATerm
    mk-new-rhs-unsafe(ATerm)                      → ATerm
    unsafeeq-conds(ATerm, ATerm, ATerm)           → ATerm
    store-par-att-conds(ATerm, ATerm, INT)        → ATerm

    mk-store-fun-att-cond(ATerm, ATerm, ATerm)    → ATerm
    mk-store-par-att-cond(ATerm, ATerm)           → ATerm
    mk-store-att(ATerm)                           → ATerm
```

equations
The function unsafeeq defined below creates a new left-hand side and a new right hand side of the equations. The conditions in the equation are transformed by means of the function unsafeeq-conds.

$$\frac{\begin{array}{r}L_1{}' = \text{scImplies},\\ T_1{}'' = \text{mk-new-lhs-eq}([T_1\ T_1{}']),\\ T_2{}' = \text{mk-new-rhs-unsafe}(\text{sort-of}(T_1)),\\ T_3{}'' = \text{unsafeeq-conds}([T_1\ T_1{}'],\ T_3{}',\ T_2)\end{array}}{\begin{array}{l}\text{unsafeeq}([\text{scCondEquation}\ L_1;\ L_2;\ [T_1\ T_1{}'];\ T_2;\ [T_3\ T_3{}']]) =\\ [\text{scCondEquation}\ L_1{}';\ L_2;\ T_1{}'';\ T_2{}';\ T_3{}'']\end{array}} \quad \text{[use1]}$$

The function mk-new-lhs-eq defined below returns an ATerm representing the new left-hand side in an incremental equation. The attribute incremental is removed by means of the function rm-inc-att and the incremental information is added as an argument by means of the function addINCINFO.

$$\frac{\begin{array}{r}T_1{}' = \text{addINCINFO}(T_1),\\ T_3{}' = \text{rm-inc-att}(T_3),\\ T_4{}' = \text{merge}(T_4,\ \text{rank-level-R-L}\ (\ ))\end{array}}{\begin{array}{l}\text{mk-new-lhs-eq}([[\text{scCfFunction}\ T_1;\ T_2;\ [L\ T_3]]\ T_4]) =\\ [[\text{scCfFunction}\ T_1{}';\ T_2;\ [L\ T_3{}']]\ T_4{}']\end{array}} \quad \text{[NlhsE]}$$

The function mk-new-rhs-unsafe defined below returns an ATerm representing the right-hand side in an *unsafe* incremental equation. T represents the result sort of the top operator on the left-hand side of the original equation.

$$\text{mk-new-rhs-unsafe}(T) = \qquad [\text{NrhsU}]$$
$$[\text{nf2sort}(T)\ \text{var-term}(\texttt{"Nf"}, \text{scNORMAL-FORM})]$$

The function unsafeeq-conds defined below creates the conditions in an incremental unsafe equation. Input: T_1 represents the left-hand side of the original equation, T_2 is a list (possibly the empty list nil) of the conditions in the original equation, and T_3 is the ATerm representing the right-hand side of the original equation.

$$\frac{\begin{array}{l} T_4 = \text{mk-decl}(T_1), \\ T_5 = \text{mk-status-cond}(\text{scUnSafe}, \text{fun-att}(T_4)), \\ T_6 = \text{store-par-att-conds}(\text{parameters}([T_1\ \ T_1{}']),\ T_4,\ 1), \\ T_7 = \text{mk-store-fun-att-cond}(\text{fun-att}(T_4),\ T_3,\ \text{sort-of}(T_1)), \\ T_8 = \text{merge}(\text{merge}(T_5,\ T_6),\ \text{merge}(T_2,\ T_7)) \end{array}}{\begin{array}{l} \text{unsafeeq-conds}([T_1\ \ T_1{}'],\ T_2,\ T_3) = \\ [\text{scConditions}\ \ T_8] \end{array}} \qquad [\text{USC}]$$

Below the function store-par-att-conds is defined. When this function is applied, conditions for storing each parameter in the left-hand side of the original equation are created.

$$\frac{\begin{array}{l} T_4 = \text{mk-store-par-att-cond}(\text{par-att}(T_3,\ \mathit{Int}),\ T_1), \\ T_5 = \text{store-par-att-conds}(T_2,\ T_3,\ \mathit{Int} + 1) \end{array}}{\text{store-par-att-conds}(T_1;\ T_2,\ T_3,\ \mathit{Int}) = \text{merge}(T_4,\ T_5)} \qquad [\text{SPAC1}]$$

$$\frac{\begin{array}{l} \text{false} = \text{isList}(T_1), \\ T_1 \neq \text{nil}, \\ T_4 = \text{mk-store-par-att-cond}(\text{par-att}(T_3,\ \mathit{Int}),\ T_1) \end{array}}{\text{store-par-att-conds}(T_1,\ T_3,\ \mathit{Int}) = T_4} \qquad [\text{SPAC2}]$$

$$\text{store-par-att-conds}(T_1,\ T_3,\ \mathit{Int}) \ = \ \text{nil} \quad \textbf{otherwise} \qquad [\text{SPAC}]$$

The function mk-store-att yields an ATerm representing the ASF+SDF term store-par-att(*AN*,| *term* |) or store-par-att(*AN*,| *term* |) where *AN* is the name of the attribute in which the the normal form | *term* | should be stored.

$$\text{mk-store-att}(T) = \qquad [\text{MSA}]$$
$$\begin{array}{l} [\text{scCfFunction} \\ \ [\text{scCfElems} \\ \ \ T;\ \texttt{"("};\ [\text{scSort scATT-NAME}];\ \texttt{","};\ [\text{scSort scNORMAL-FORM}];\ \texttt{")"}]; \\ \ [\text{scSort scNORMAL-FORM}]; \\ \ [\text{scAttributes nil}]] \end{array}$$

The function mk-store-fun-att-cond returns an ATerm representing a condition in which the function store-fun-att is invoked to store a computed value in an attribute in the storage.

$$\text{mk-store-fun-att-cond}(T,\ T_1,\ T_2) = \qquad [\text{MSFA}]$$
$$\begin{array}{l} [\text{scCondition scEq}; \\ \qquad \text{var-term}(\texttt{"Nf"},\ T_2); \\ \qquad [\text{mk-store-att}(\text{scstore-fun-att}) \\ \qquad\ \text{lex-term}(T,\ \text{scATT-NAME}); \\ \qquad\ [\text{sort2nf}(\text{sort-of}(\text{operator}(T_1)))\ \ T_1]]] \end{array}$$

The function mk-store-par-att-cond returns an ATerm representing a condition in which the function store-par-att is invoked to store the value of a parameter in an attribute in the storage.

```
mk-store-par-att-cond(T, T1) =                                   [MSPA]
 [scCondition scEq;
            bool-constant(sctrue);
            [mk-store-att(scstore-par-att)
             lex-term(T, scATT-NAME);
             [sort2nf(sort-of(T1)) T1]]]
```

G.2.8 StatusCondition

In the module StatusCondition the specification is given for the generation or a status-is-safe of status-is-unsafe condition.

module StatusCondition
imports AuxMod2IncMod$^{G.2.9}$
exports
  context-free syntax
    mk-status-cond(ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    mk-status-fun(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
equations

The function mk-status-cond defined below returns an ATerm representing the condition true = status-is-safe(*AN*,*IncInfo*) or true = status-is-unsafe(*AN*,*R*:*L*) depending on the string constants scSafe and scUnSafe, variable T_1.. T_2 represents the name of the attributes which status should be checked.

```
mk-status-cond(T1, T2) =                                        [SCond1]
 [scCondition scEq;
            bool-constant(sctrue);
            mk-status-fun(T1, T2, rank-level-R-L ( ))]
```

The function mk-status-fun yields an ATerm representing the ASF+SDF term status-is-safe or status-is-unsafe with arguments.

```
mk-status-fun(T1, T2, T3) =                                       [MSF]
 [[scCfFunction
   [scCfElems
    T1; "("; [scSort scATT-NAME]; ","; [scSort scINCINFO]; ")"];
   [scSort scBOOL];
   [scAttributes nil]]
  lex-term(T2, scATT-NAME);
  T3]
```

G.2.9 AuxMod2IncMod

In the module AuxMod2IncMod some auxiliary functions have been defined.

module AuxMod2IncMod
imports AuxIncModule$^{G.1.3}$

exports

context-free syntax

```
addINCINFO(ATerm) → ATerm
nf2sort(ATerm)    → ATerm
```

equations

Below the function addINCINFO is defined. This function adds the sort INCINFO as an argument in the function declaration of a function *T* by means of the function add2cfelems.

```
addINCINFO(T) = add2cfelems(T, [scSort scINCINFO])          [AII]
```

The function nf2sort defined below yields an ATerm aterm representing the declaration of a conversion function: nf2SORT(NORMAL-FORM) →SORT.

```
nf2sort(T) =                                                [N2S]
 [scCfFunction
  [scCfElems
   cc("nf2"; T); "("; [scSort scNORMAL-FORM]; ")"];
  [scSort T];
  [scAttributes nil]]
```

G.3 Transform module, ASF+SDF version

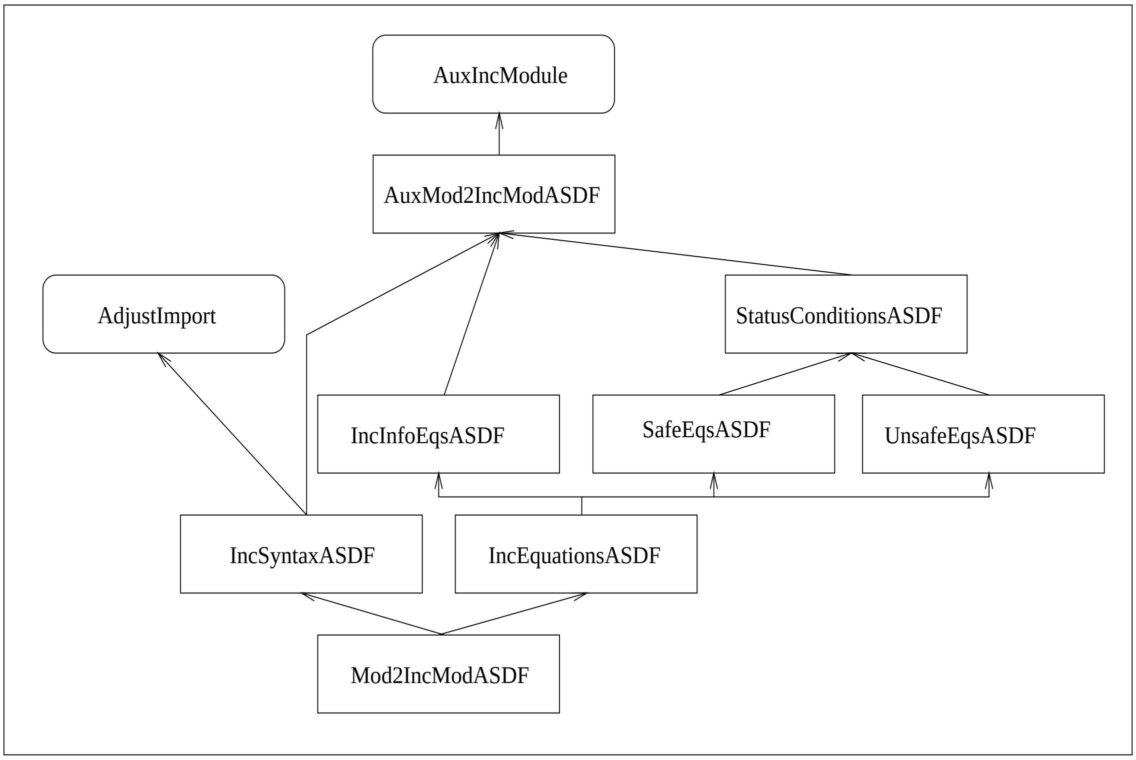


Figure G.3: Module import overview.

G.3.1 Mod2IncModASDF

In the module Mod2IncModASDF the transformation of a module into an incremental module is specified. The result of this transformation can be used when the storage for incremental reduction is passed as an argument in the incremental function and is part of the result returned by an incremental function.

module Mod2IncModASDF
imports IncSyntaxASDF$^{G.3.2}$ IncEquationsASDF$^{G.3.3}$
exports
  context-free syntax
    mod2incmodASDF(ATerm, ATerm) $\rightarrow$ ATerm
    mod2incmodASDF(ATerm) $\rightarrow$ ATerm
equations
The function mod2incmodASDF defined below transforms a module into an incremental module. This means that functions that are defined as incremental and the equations defined for these functions are transformed in such a way that when terms are applied to the transformed equations, they can be reduced incrementally. The input ATerm represents a module in ASFIX notation and the input ATerm T_5 contains a list of the names of the other modules that are transformed. We need this information to be able to change the names of imported modules that are transformed or are going to be transformed. In the first condition, all declarations of functions declared as incremental in the module are selected by means of the function inc-operators. The sorts for which new definitions of the sort NORMAL-FORM are selected by means of function conversion-sorts. At this point we are also interested in the sorts of the constructors, the first arguments in incremental functions. These sorts are selected when the function constructor-sorts is applied. In the fourth

condition the name of the module is changed by adding the prefix `"Inc"`. The source syntax and the source equations are used to restore comment. When a module in AsFix-representation has been changed, the comments can no longer be restored and these two parts are given the new value nil. The syntax section and the equations in the module are transformed by means of the functions syn2incsyn and eqs2inceqs respectively. The equation [MIE2] applies when no list of names of other transformed is supplied.

$$\frac{\begin{array}{l} T_6 = \text{inc-operators}(\text{cf-syntax}(T_2)), \\ T_7 = \text{conversion-sorts}(T_6), \\ T_8 = \text{constructor-sorts}(T_6), \\ L_1{}' = \text{cc}(\texttt{"Inc"};\ L_1), \\ T_1{}' = [\text{scSourceSyntax nil}], \\ T_2{}' = \text{syn2incsyn}(T_2,\ T_5,\ T_7,\ T_8), \\ T_3{}' = [\text{scSourceEquations nil}], \\ T_4{}' = \text{eqs2inceqs}(T_4,\ T_6) \end{array}}{\begin{array}{l} \text{mod2incmodASDF}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4],\ T_5) = \\ [\text{scModule } L_1{}';\ L_2;\ T_1{}';\ T_2{}';\ T_3{}';\ T_4{}'] \end{array}} \quad \text{[MIE1]}$$

$$\text{mod2incmodASDF}(T) \ = \ \text{mod2incmodASDF}(T, \texttt{""}) \quad \text{[MIE2]}$$

G.3.2 IncSyntaxASDF

In this module the transformation of the syntax section of a module for the Asf+Sdf version is given.

module IncSyntaxASDF
imports AdjustImport$^{G.2.3}$ AuxMod2IncModASDF$^{G.3.8}$
exports
  context-free syntax
    syn2incsyn(ATerm, ATerm, ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    trans-fun-decls(ATerm) → ATerm
equations

The function syn2incsyn defined below transforms the syntax section in a module into an incremental syntax section. The input ATerm T contains the syntax sections of a module in AsFix notation, T_1 contains the names of other modules that have been, or shall be, transformed. T_2 contains the sorts for which new NORMAL-FORMs must be declared. T_3 contain a list of the sorts of the constructors that occur in the module. First, the import section is adjusted by means of function adjust-import. This function yields a new copy of the syntax section. The declarations of the functions that have been declared as incremental are transformed when the function trans-fun-decls is applied. Declarations of new NORMAL-FORMs are created when the function mk-new-normalforms is applied. Variables are defined for all the conversion sorts and the sorts of the constructors in the fourth condition by means of the the function mk-numbered-variables. The name of the module in which the functions for manipulating the storage are defined is added to the import section in the module by means of the function add2imports. Finally, the declarations of the new NORMAL-FORMs and the variables are added to a context-free syntax section and a variable section in a hidden part of the module. This is done when the function add2hiddens is applied. If no hidden section is present in the module, one will be created.

$$\frac{\begin{array}{l} T' = \text{adjust-import}(T,\ T_1), \\ T'' = \text{trans-fun-decls}(T'), \\ T_4 = \text{mk-new-normalforms}(T_2), \\ T_5 = \text{mk-numbered-variables}(T_2), \\ T_6 = \text{mk-variables}(T_3), \\ T_7 = \text{add2imports}(T'',\ [\text{scId}\ \texttt{"SM-during-reduction"}]), \\ T_8 = \text{add2hiddens}(T_7,\ T_4,\ \text{merge}(T_5,\ T_6)) \end{array}}{\begin{array}{l} \text{syn2incsyn}([\text{scPrefixSyntax}\ T],\ T_1,\ T_2,\ T_3) = \\ [\text{scPrefixSyntax}\ T_8] \end{array}} \quad \text{[SIS1]}$$

The function trans-fun-decls defined below transforms the declarations of funtions that have been declared as incremental. In equation [TFD6] the attribute incremental is removed when the function rm-inc-att is applied to the attributes of the function. A copy is made of the function declaration and to this copy the sorts INCINFO and STORAGE are added as new arguments. The STORAGE is added to the original.

$$\text{trans-fun-decls}(T_1;\ T_2) = \text{merge}(\text{trans-fun-decls}(T_1),\ \text{trans-fun-decls}(T_2)) \quad \text{[TFD1]}$$

$$\text{trans-fun-decls}([\text{scImports}\ T]) = [\text{scImports}\ T] \quad \text{[TDF2]}$$

$$\text{trans-fun-decls}([\text{scExports}\ T]) = [\text{scExports trans-fun-decls}(T)] \quad \text{[TDF3]}$$

$$\text{trans-fun-decls}([\text{scHiddens}\ T]) = [\text{scHiddens trans-fun-decls}(T)] \quad \text{[TDF4]}$$

$$\text{trans-fun-decls}([\text{scContextFreeSyntax}\ T]) = [\text{scContextFreeSyntax trans-fun-decls}(T)] \quad \text{[TFD5]}$$

$$\frac{\begin{array}{l} \text{true} = \text{isInc}(T_3), \\ {T_1}' = \text{addSTORAGE}(T_1), \\ {T_1}'' = \text{addINCINFO-STORAGE}(T_1), \\ {T_2}' = [\text{scSort scNF-STORAGE}], \\ {T_3}' = \text{rm-inc-att}(T_3) \end{array}}{\begin{array}{l} \text{trans-fun-decls}([\text{scCfFunction}\ T_1;\ T_2;\ [L\ T_3]]) = \\ [\text{scCfFunction}\ {T_1}';\ {T_2}';\ [L\ {T_3}']];\ [\text{scCfFunction}\ {T_1}'';\ {T_2}';\ [L\ {T_3}']] \end{array}} \quad \text{[TFD6]}$$

$$\text{trans-fun-decls}(T) = T \quad \textbf{otherwise} \quad \text{[TFD]}$$

G.3.3 IncEquationsASDF

In the module IncEquations the transformation of equations for the ASF+SDF version is specified.

module IncEquationsASDF
imports IncInfoEqsASDF$^{G.3.4}$ SafeEqsASDF$^{G.3.5}$ UnsafeEqsASDF$^{G.3.6}$
exports
  context-free syntax
    eqs2inceqs(ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    trans-eqs(ATerm) $\rightarrow$ ATerm

equations

The function eqs2inceqs defined below transforms the equations in the equation section of a module. T is a list of equations, T_1 is a list of the declaration of the functions that were declared as incremental in the syntax section of the module being transformed. In the first condition *indicating* equations are created by means of the function inc-info-eqs. In the second condition equations are created for the case that the value of a related attribute is reliable. In the last condition the existing equations in the prefix syntax are transformed.

$$\frac{\begin{array}{l} T_3 = \text{inc-info-eqs}(T_1,\ T,\ 1), \\ T_6 = \text{safeeqs}(T_1,\ T,\ 1), \\ T_4 = \text{trans-eqs}(T) \end{array}}{\begin{array}{l} \text{eqs2inceqs}([\text{scPrefixEquations}\ T],\ T_1) = \\ [\text{scPrefixEquations merge}(T_3,\ \text{merge}(T_6,\ T_4))] \end{array}} \quad \text{[E2IE]}$$

The function trans-eqs defined below transforms all equations in the module. Equations which top operator has not been declared as incremental are left intact. If the top operator in an equation has been declared as incremental the equation is transformed into an equation which first argument invokes a function status-is-unsafe during reduction. Before the unsafe equation can be created by means of the function unsafeeq, the rank and the level is added to the left-hand side of the equation and the incremental terms in the conditions and on the right-hand side of the equation by means of the function add-ranks-level.

$$\text{trans-eqs}(T_1;\ T_2) \ = \ \text{merge}(\text{trans-eqs}(T_1), \text{trans-eqs}(T_2)) \quad \text{[TEq1]}$$

$$\frac{\begin{array}{l} \text{true} = \text{isInc}(\text{operator}(T_1)), \\ T_4 = \text{unsafeeq}(\text{add-ranks-level}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ \text{rank-table}(T_1))) \end{array}}{\text{trans-eqs}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = T_4} \quad \text{[TEq2]}$$

$$\text{trans-eqs}(T) \ = \ T \quad \textbf{otherwise} \quad \text{[TEq]}$$

G.3.4 IncInfoEqsASDF

In the module IncInfoEqsASDF the generation of equations introducing the incremental information has been given.

module IncInfoEqsASDF
imports AuxMod2IncModASDF$^{G.3.8}$
exports
  context-free syntax
    inc-info-eqs(ATerm, ATerm, INT) $\rightarrow$ ATerm
equations

Below the function inc-info-eqs is defined. This function has two arguments. The first argument is a list of all function declarations of functions in the module that is being transformed that have been declared as incremental. The second argument is a list containing all the equations in the module. For each function declaration one defining equation is selected. The left-hand side of the selected equation is kept an the rest of the equation wasted. A copy is made of this left-hand side and the storage S is added to the original and the incremental information, the INCINFO, and the storage S is added to the copy. The INCINFO is represented in the copy by *1:0*, thus the rank is 1 and the level is 0. The copy is made the right-hand side of the equation, thus when reduction starts, the rank and the level in the term is initialised with the values 1 and 0 respectively. The third argument in the function inc-info-eqs is used to number these equations.

$$\text{inc-info-eqs}(T_1;\ T_2,\ T_3,\ \mathit{Int}) = \qquad \text{[IIEq1]}$$
$$\text{merge}(\text{inc-info-eqs}(T_1,\ T_3,\ \mathit{Int}),\ \text{inc-info-eqs}(T_2,\ T_3,\ \mathit{Int}+1))$$

$$\text{inc-info-eqs}(T,\ [\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) = \qquad \text{[IIEq2]}$$
$$\text{inc-info-eqs}(T,\ [\text{scCondEquation } L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3],\ \mathit{Int})$$

$$\text{inc-info-eqs}(T,\ [\text{scCondEquation } L_1;\ L_2;\ [T_1\ T_1{}'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) \qquad \text{[IIEq3]}$$
$$= \text{inc-info-eqs}(T,\ T_4,\ \mathit{Int})$$
when
$$T \neq T_1$$

$$\frac{\begin{array}{r@{\ }l} T_1{}' &= \text{addSTORAGE}(T_1), \\ T_1{}'' &= \text{addINCINFO-STORAGE}(T_1), \\ T_2{}' &= [\text{scSort scNF-STORAGE}], \\ T_7 &= \text{sort2var}(\text{first-sort}(T_1)), \\ T_4{}' &= \text{lex-term}(\texttt{"1:0"},\ \text{scINCINFO}), \\ T_9{}' &= \text{varS ( )}, \\ T_9{}'' &= \text{merge}(T_4{}',\ \text{varS ( )}), \\ T_6{}' &= [\text{scConditions nil}], \\ L_3{}' &= \text{tag}(\texttt{"AII"},\ \mathit{Int}) \end{array}}{\begin{array}{l}\text{inc-info-eqs}([L\ T_1;\ T_2;\ T_3],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ T_3]\ T_8;\ T_9];\ T_5;\ T_6],\ \mathit{Int}) = \\ [L_1\ L_2;\ L_3{}';\ [[L\ T_1{}';\ T_2{}';\ T_3]\ T_7;\ T_9{}'];\ [[L\ T_1{}'';\ T_2{}';\ T_3]\ T_7;\ T_9{}''];\ T_6{}']\end{array}} \qquad \text{[IIEq4]}$$

$$\frac{\begin{array}{r@{\ }l} T_1{}' &= \text{addSTORAGE}(T_1), \\ T_1{}'' &= \text{addINCINFO-STORAGE}(T_1), \\ T_2{}' &= [\text{scSort scNF-STORAGE}], \\ T_7 &= \text{sort2var}(\text{first-sort}(T_1)), \\ T_4{}' &= \text{lex-term}(\texttt{"1:0"},\ \text{scINCINFO}), \\ T_{10} &= \text{varS ( )}, \\ T_6{}' &= [\text{scConditions nil}], \\ L_3{}' &= \text{tag}(\texttt{"AII"},\ \mathit{Int}) \end{array}}{\begin{array}{l}\text{inc-info-eqs}([L\ T_1;\ T_2;\ T_3],\ [L_1\ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ T_3]\ [T_8\ T_9]];\ T_5;\ T_6],\ \mathit{Int}) = \\ [L_1\ L_2;\ L_3{}';\ [[L\ T_1{}';\ T_2{}';\ T_3]\ T_7;\ T_{10}];\ [[L\ T_1{}'';\ T_2{}';\ T_3]\ T_7;\ T_4{}';\ T_{10}];\ T_6{}']\end{array}} \qquad \text{[IIEq4]}$$

$$\text{inc-info-eqs}(T,\ T_1,\ \mathit{Int}) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[IIEq]}$$

G.3.5 SafeEqsASDF

In the module SafeEqsASDF the generation of equations for the safe case has been specified.

module SafeEqsASDF
imports StatusConditionASDF$^{G.3.7}$
exports
  context-free syntax
    safeeqs(ATerm, ATerm, INT) $\rightarrow$ ATerm
hiddens
  context-free syntax
    mk-new-rhs-safe(ATerm) $\rightarrow$ ATerm
equations

The function safeeqs defined below is similar to the function inc-info-eqs and has the same arguments. For each function declaration one defining equation is selected. In the left-hand side of this equation the variables representing the constructor are replaced by a new variable and the INCINFO and the STORAGE are added as arguments. A condition is created by means of the function mk-status-cond. In this condition a function will be invoked during reduction to check that the value of the related function attribute is reliable (safe). A new right-hand side of the equation is created in which the function retrieve will be invoked during reduction of a term to retrieve the value from the storage. This is accomplished when the function mk-new-rhs-safe is applied. Equation [SEq4] applies when the term on left-hand side of the equation has parameters and when there are no parameters in this term, equation [SEq5] applies.

$$\mathsf{safeeqs}(T_1;\ T_2,\ T_3,\ \mathit{Int}) \ = \ \mathsf{merge}(\mathsf{safeeqs}(T_1,\ T_3,\ \mathit{Int}), \mathsf{safeeqs}(T_2,\ T_3,\ \mathit{Int}+1)) \qquad \text{[SEq1]}$$

$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) = \qquad \text{[SEq2]}$$
$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ T'];\ T_2;\ T_3],\ \mathit{Int})$$

$$\mathsf{safeeqs}(T,\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ [T_1\ \ T_1{}'];\ T_2;\ T_3];\ T_4,\ \mathit{Int}) \qquad \text{[SEq3]}$$
$$= \mathsf{safeeqs}(T,\ T_4,\ \mathit{Int})$$
$$\textbf{when}$$
$$T \neq T_1$$

$$\frac{\begin{array}{l} L_2{}' = \mathsf{scImplies}, \\ L_3{}' = \mathsf{tag}(\texttt{"default-SIS"},\ \mathit{Int}), \\ T_1{}' = \mathsf{addINCINFO\text{-}STORAGE}(T_1), \\ T_2{}' = [\mathsf{scSort\ scNF\text{-}STORAGE}], \\ T_3{}' = \mathsf{rm\text{-}inc\text{-}att}(T_3), \\ T_4{}' = \mathsf{sort2var}(\mathsf{first\text{-}sort}(T_1)), \\ T_9 = \mathsf{rank\text{-}level\text{-}R\text{-}L}\ (\ ), \\ T_{10} = \mathsf{varS}\ (\ ), \\ T_5{}' = \mathsf{merge}(T_5,\ T_9;\ T_{10}), \\ T_8 = \mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([L\ T_1;\ T_2;\ T_3])), \\ T_6{}' = \mathsf{mk\text{-}new\text{-}rhs\text{-}safe}(T_8), \\ T_7{}' = [\mathsf{scConditions\ mk\text{-}status\text{-}cond}(T_8,\ \mathsf{scSafe})] \end{array}}{\begin{array}{l} \mathsf{safeeqs}([L\ T_1;\ T_2;\ [L_4\ \ T_3]],\ [L_1\ \ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ [L_4\ \ T_3]]\ T_4;\ T_5];\ T_6;\ T_7],\ \mathit{Int}) = \\ [L_1\ \ L_2{}';\ L_3{}';\ [[L\ T_1{}';\ T_2{}';\ [L_4\ \ T_3{}']]\ T_4{}';\ T_5{}'];\ T_6{}';\ T_7{}'] \end{array}} \qquad \text{[SEq4]}$$

$$\frac{\begin{array}{l} L_2{}' = \mathsf{scImplies}, \\ L_3{}' = \mathsf{tag}(\texttt{"default-SIS"},\ \mathit{Int}), \\ T_1{}' = \mathsf{addINCINFO\text{-}STORAGE}(T_1), \\ T_2{}' = [\mathsf{scSort\ scNF\text{-}STORAGE}], \\ T_3{}' = \mathsf{rm\text{-}inc\text{-}att}(T_3), \\ T_4{}' = \mathsf{sort2var}(\mathsf{first\text{-}sort}(T_1)), \\ T_9 = \mathsf{rank\text{-}level\text{-}R\text{-}L}\ (\ ), \\ T_{10} = \mathsf{varS}\ (\ ), \\ T_8 = \mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([L\ T_1;\ T_2;\ T_3])), \\ T_6{}' = \mathsf{mk\text{-}new\text{-}rhs\text{-}safe}(T_8), \\ T_7{}' = [\mathsf{scConditions\ mk\text{-}status\text{-}cond}(T_8,\ \mathsf{scSafe})] \end{array}}{\begin{array}{l} \mathsf{safeeqs}([L\ T_1;\ T_2;\ [L_4\ \ T_3]],\ [L_1\ \ L_2;\ L_3;\ [[L\ T_1;\ T_2;\ [L_4\ \ T_3]]\ [T_4\ \ T_5]];\ T_6;\ T_7],\ \mathit{Int}) = \\ [L_1\ \ L_2{}';\ L_3{}';\ [[L\ T_1{}';\ T_2{}';\ [L_4\ \ T_3{}']]\ T_4{}';\ T_9;\ T_{10}];\ T_6{}';\ T_7{}'] \end{array}} \qquad \text{[SEq5]}$$

$$\mathsf{safeeqs}(T,\ T_1,\ \mathit{Int}) \ = \ \mathsf{nil} \quad \textbf{otherwise} \qquad \text{[SEq]}$$

The function mk-new-rhs-safe defined below returns an ATerm representing the right-hand side in a *safe* incremental equation. *T* represents the function attribute related to the original equation.

```
mk-new-rhs-safe(T) =                                                   [NrhsS]
 [[scCfFunction
   [scCfElems
    scretrieve; "("; [scSort scATT-NAME]; ","; [scSort scSTORAGE]; ")"];
   [scSort scNF-STORAGE];
   [scAttributes nil]]
  lex-term(T, scATT-NAME);
  varS("1")]
```

G.3.6 UnsafeEqsASDF

In the module UnsafeEqASDF the generation of an incremental function for the case when the value in the attribute is not reliable for the ASF+SDF version.

module UnsafeEqsASDF

```
imports StatusConditionASDF^G.3.7
exports
  context-free syntax
    unsafeeq(ATerm) → ATerm
hiddens
  context-free syntax
    store-fun-att-conds(ATerm, ATerm, ATerm, INT) → ATerm
    trans-cond(ATerm, ATerm, ATerm)               → ATerm
    trans-pars(ATerm, ATerm, ATerm)               → ATerm
    trans-terms(ATerm, ATerm, ATerm)              → ATerm

    mk-new-lhs-eq(ATerm)                          → ATerm
    new-rhs(ATerm, ATerm, ATerm)                  → ATerm
    mk-store-fun-att-cond(ATerm, ATerm, ATerm)    → ATerm
    mk-store-par-att-cond(ATerm, ATerm, ATerm)    → ATerm
    mk-store-att(ATerm)                           → ATerm
    new-cond(ATerm, ATerm, ATerm)                 → ATerm
    lhs-inc-cond(ATerm, ATerm)                    → ATerm
  variables
    Cond [0-9']*   → ATerm
    Conds [0-9']*  → ATerm
    Var [0-9']*    → ATerm
    Terms [0-9']*  → ATerm
    VarTab [0-9']* → ATerm
    Svar [0-9']*   → ATerm
    Term [0-9']*   → ATerm
    Pars [0-9']*   → ATerm
equations
```

Below the definition is given of the function unsafeeq. The input is an equation in ASFIX representation. Equation [USEq1] applies when the left-hand side has parameters, otherwise equation [USEq2]. First a string representation of the functions declarations of the left-hand side of the equation is made, *L* and the left-hand side is created. A condition in which the status of the attribute is checked is generated when the function mk-status-cond is generated. Conditions in which the values of the parameters are stored are created when the function store-fun-att-conds.

The conditions in the original equation are transformed (trans-cond) and the right-side of the equation is transformed by applying trans-pars. When these conditions are generated, an annotated result is returned. *Cond* are the conditions, *Svar* is the index of the variable *S*, *VarTab* is a table with the indices of variables that have been introduced during the transformation and the *Term* is the term which value should be stored as result of the computation.

$$\frac{\begin{array}{r@{\;}l} L_1{}' &= \mathsf{scImplies}, \\ L &= \mathsf{mk\text{-}decl}(T), \\ T_1{}' &= \mathsf{mk\text{-}new\text{-}lhs\text{-}eq}([T\ T';\ T'']), \\ Cond_1 &= \mathsf{mk\text{-}status\text{-}cond}(\mathsf{fun\text{-}att}(L),\ \mathsf{scUnSafe}), \\ Conds_1\ /\ Svar &= \mathsf{store\text{-}fun\text{-}att\text{-}conds}(T'',\ L,\ \texttt{"1"},\ 1), \\ Conds_2\ /\ VarTab_1\ /\ Svar_1 &= \mathsf{trans\text{-}cond}(T_3,\ \mathsf{nil},\ Svar), \\ Conds_3\ /\ Term\ /\ VarTab_2\ /\ Svar_2 &= \mathsf{trans\text{-}pars}(T_2,\ VarTab_1,\ Svar_1), \\ T_2{}' &= \mathsf{new\text{-}rhs}(\mathsf{fun\text{-}att}(L),\ Term,\ Svar_2), \\ T_4 &= \mathsf{merge}(Cond_1,\ Conds_1), \\ T_5 &= \mathsf{merge}(Conds_2,\ Conds_3), \\ T_3{}' &= \mathsf{merge}(T_4,\ T_5) \end{array}}{\begin{array}{l} \mathsf{unsafeeq}([\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ T';\ T''];\ T_2;\ [L_3\ T_3]]) = \\ [\mathsf{scCondEquation}\ L_1{}';\ L_2;\ T_1{}';\ T_2{}';\ [L_3\ T_3{}']] \end{array}} \quad \text{[USEq1]}$$

$$\frac{\begin{array}{r@{\;}l} L_1{}' &= \mathsf{scImplies}, \\ L &= \mathsf{mk\text{-}decl}(T), \\ T_1{}' &= \mathsf{mk\text{-}new\text{-}lhs\text{-}eq}([T\ [T'\ T'']]), \\ Cond_1 &= \mathsf{mk\text{-}status\text{-}cond}(\mathsf{fun\text{-}att}(L),\ \mathsf{scUnSafe}), \\ Conds_2\ /\ VarTab_1\ /\ Svar_1 &= \mathsf{trans\text{-}cond}(T_3,\ \mathsf{nil},\ \texttt{"1"}), \\ Conds_3\ /\ Term\ /\ VarTab_2\ /\ Svar_2 &= \mathsf{trans\text{-}pars}(T_2,\ VarTab_1,\ Svar_1), \\ T_2{}' &= \mathsf{new\text{-}rhs}(\mathsf{fun\text{-}att}(L),\ Term,\ Svar_2), \\ T_3{}' &= \mathsf{merge}(Cond_1,\ \mathsf{merge}(Conds_2,\ Conds_3)) \end{array}}{\begin{array}{l} \mathsf{unsafeeq}([\mathsf{scCondEquation}\ L_1;\ L_2;\ [T\ [T'\ T'']];\ T_2;\ [L_3\ T_3]]) = \\ [\mathsf{scCondEquation}\ L_1{}';\ L_2;\ T_1{}';\ T_2{}';\ [L_3\ T_3{}']] \end{array}} \quad \text{[USEq2]}$$

The function store-fun-att-conds defined below creates a condition in which each parameter term is stored.

$$\frac{\begin{array}{l} Cond\ /\ Svar_1 = \mathsf{store\text{-}fun\text{-}att\text{-}conds}(T_1,\ T,\ Svar,\ Int), \\ Conds\ /\ Svar_2 = \mathsf{store\text{-}fun\text{-}att\text{-}conds}(T_1;\ T_2,\ T,\ Svar_1,\ Int+1), \\ \quad Conds' = \mathsf{merge}(Cond,\ Conds) \end{array}}{\begin{array}{l} \mathsf{store\text{-}fun\text{-}att\text{-}conds}(T_1;\ T_2,\ T,\ Svar,\ Int) = \\ Conds'\ /\ Svar_2 \end{array}} \quad \text{[SFACs1]}$$

$$\frac{\begin{array}{l} T_2 = \mathsf{par\text{-}att}(T_1,\ Int), \\ Cond = \mathsf{mk\text{-}store\text{-}par\text{-}att\text{-}cond}(T_2,\ [T\ T'],\ Svar), \\ Svar_1 = \mathsf{next\text{-}var\text{-}index}(Svar) \end{array}}{\begin{array}{l} \mathsf{store\text{-}fun\text{-}att\text{-}conds}([T\ T'],\ T_1,\ Svar,\ Int) = \\ Cond\ /\ Svar_1 \end{array}} \quad \text{[SFACs2]}$$

$$\mathsf{store\text{-}fun\text{-}att\text{-}conds}(T,\ T_1,\ Svar,\ Int)\ =\ \mathsf{nil}\ /\ Svar \quad \textbf{otherwise} \qquad \text{[SFACs]}$$

The function trans-cond defined below transformes the conditions in the original equaition. Equation [TCond1] applies when both sides of the condition have incremental top operators and are both transformed by means of the function trans-pars. This function introduces new variables for the incremental terms transformed, and these variables are returned; Var_1 and Var_2. With these two variables a new conditions is made. Conditions for the original terms where created when trans-pars was applied.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{true} & \mathsf{isInc}(\mathsf{operator}(T_1)),\\ \mathsf{true} & \mathsf{isInc}(\mathsf{operator}(T_2)),\\ \mathit{Conds}_1 \;/\; \mathit{Var}_1 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 & \mathsf{trans\text{-}pars}(T_1, \mathit{VarTab}, \mathit{Svar}),\\ \mathit{Conds}_2 \;/\; \mathit{Var}_2 \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 & \mathsf{trans\text{-}pars}(T_2, \mathit{VarTab}_1, \mathit{Svar}_1),\\ \mathit{Conds} & \mathsf{merge}(\mathit{Conds}_1, \mathit{Conds}_2),\\ \mathit{Cond} & [\mathsf{scCondition}\ L;\ \mathit{Var}_1;\ \mathit{Var}_2] \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ L;\ T_1;\ T_2], \mathit{VarTab}, \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds}, \mathit{Cond}) \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 \end{array}} \quad [\mathsf{TCond1}]$$

Equation [TCond2] applies when the left-hand side of a condition is incremental and the other side is not. We let them change places.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{true} & \mathsf{isInc}(\mathsf{operator}(T_1)),\\ \mathsf{false} & \mathsf{isInc}(\mathsf{operator}(T_2)) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ L;\ T_1;\ T_2], \mathit{VarTab}, \mathit{Svar}) =\\ \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ L;\ T_2;\ T_1], \mathit{VarTab}, \mathit{Svar}) \end{array}} \quad [\mathsf{TCond2}]$$

Equation [TCond3]: When the righ-hand side of an equation is incremental and the other not, and it is an the condition is an equality (scEq), there is no need to introduce a new condition. The right- and left- hand sides are separately transformed by means of the function trans-term. These applications will return two terms, Term_1 and Term_1. With these two terms a new condition is created by means of the function new-cond.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{false} & \mathsf{isInc}(\mathsf{operator}(T_1)),\\ \mathsf{true} & \mathsf{isInc}(\mathsf{operator}(T_2)),\\ \mathit{Conds}_1 \;/\; \mathit{Term}_1 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 & \mathsf{trans\text{-}terms}(T_1, \mathit{VarTab}, \mathit{Svar}),\\ \mathit{Conds}_2 \;/\; \mathit{Term}_2 \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 & \mathsf{trans\text{-}terms}(T_2, \mathit{VarTab}_1, \mathit{Svar}_1),\\ \mathit{Conds}_3 & \mathsf{merge}(\mathit{Conds}_1, \mathit{Conds}_2),\\ \mathit{Cond} & \mathsf{new\text{-}cond}(\mathit{Term}_1, \mathit{Svar}_2, \mathit{Term}_2),\\ \mathit{Svar}_3 & \mathsf{next\text{-}var\text{-}index}(\mathit{Svar}_2) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ \mathsf{scEq};\ T_1;\ T_2], \mathit{VarTab}, \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds}_3, \mathit{Cond}) \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_3 \end{array}} \quad [\mathsf{TCond3}]$$

Equation [TCond4]: When the condition is an inequality and the right-hand side is an incremental term a new conditions must be introduced for the incremental term. This is done when the function trans-pars is applied. The function trans-pars returns an introdued variable Var_2 that represents the the incremental term and can be placed in the inequality.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{false} & \mathsf{isInc}(\mathsf{operator}(T_1)),\\ \mathsf{true} & \mathsf{isInc}(\mathsf{operator}(T_2)),\\ \mathit{Conds}_1 \;/\; \mathit{Term}_1 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 & \mathsf{trans\text{-}terms}(T_1, \mathit{VarTab}, \mathit{Svar}),\\ \mathit{Conds}_2 \;/\; \mathit{Var}_2 \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 & \mathsf{trans\text{-}pars}(T_2, \mathit{VarTab}_1, \mathit{Svar}_1),\\ \mathit{Conds}_3 & \mathsf{merge}(\mathit{Conds}_1, \mathit{Conds}_2),\\ \mathit{Cond} & [\mathsf{scCondition}\ \mathsf{scNeq};\ \mathit{Var}_2;\ \mathit{Term}_1] \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ \mathsf{scEq};\ T_1;\ T_2], \mathit{VarTab}, \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds}_3, \mathit{Cond}) \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 \end{array}} \quad [\mathsf{TCond4}]$$

When both sides of a condition are non-incremental, there is no need to introduce new conditions.

The two terms are transformed by means of the function trans-terms.

$$\frac{\begin{array}{r@{\;}l} \mathsf{false} &= \mathsf{isInc}(\mathsf{operator}(T_1)),\\ \mathsf{false} &= \mathsf{isInc}(\mathsf{operator}(T_2)),\\ \mathit{Conds}_1 \;/\; \mathit{Term}_1 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 &= \mathsf{trans\text{-}terms}(T_1,\ \mathit{VarTab},\ \mathit{Svar}),\\ \mathit{Conds}_2 \;/\; \mathit{Term}_2 \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 &= \mathsf{trans\text{-}terms}(T_2,\ \mathit{VarTab}_1,\ \mathit{Svar}_1),\\ \mathit{Conds}_3 &= \mathsf{merge}(\mathit{Conds}_1,\ \mathit{Conds}_2),\\ \mathit{Cond} &= [\mathsf{scCondition}\ L;\ \mathit{Term}_1;\ \mathit{Term}_2] \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}([\mathsf{scCondition}\ L;\ T_1;\ T_2],\ \mathit{VarTab},\ \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds}_3,\ \mathit{Cond}) \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 \end{array}} \quad \text{[TCond5]}$$

Equation [TCond6] applies when a list of conditions must be transformed.

$$\frac{\begin{array}{l} \mathit{Conds}_1 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 = \mathsf{trans\text{-}cond}(T_1,\ \mathit{VarTab},\ \mathit{Svar}),\\ \mathit{Conds}_2 \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 = \mathsf{trans\text{-}cond}(T_2,\ \mathit{VarTab}_1,\ \mathit{Svar}_1) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}cond}(T_1;\ T_2,\ \mathit{VarTab},\ \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds}_1,\ \mathit{Conds}_2) \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 \end{array}} \quad \text{[TCond6]}$$

$$\mathsf{trans\text{-}cond}(T,\ \mathit{VarTab},\ \mathit{Svar}) \;=\; \mathsf{nil} \;/\; \mathit{VarTab} \;/\; \mathit{Svar} \quad \textbf{otherwise} \qquad \text{[TCond]}$$

The function trans-pars is defined below. Equation [TPars1]: The top operator of the parameter is incremental. The term has no parameters of its own. A new variable is created to be the new left-hand side of the condition that is created. The variable S representing the storage will be combined with the created variable. The variable that was created to represent the term is returned, *Var*.

$$\frac{\begin{array}{r@{\;}l} \mathsf{true} &= \mathsf{isInc}(T_3),\\ \mathit{Var} \;/\; \mathit{VarTab}_1 &= \mathsf{new\text{-}var}(\mathsf{sort\text{-}of}(T_2),\ \mathit{VarTab}),\\ \mathit{Cond} &= \mathsf{new\text{-}cond}(\mathit{Var},\ \mathit{Svar},\ [[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ [T_4\ \ T_4']]),\\ \mathit{Svar}_1 &= \mathsf{next\text{-}var\text{-}index}(\mathit{Svar}) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}pars}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ [T_4\ \ T_4']],\ \mathit{VarTab},\ \mathit{Svar}) =\\ \mathit{Cond} \;/\; \mathit{Var} \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 \end{array}} \quad \text{[TPars1]}$$

When the parameter has parameters of its own, these parameters are first transformed. The transformation of its parameters yields terms, *Terms*. If the parameters had incremental top operators the returned terms will be introduced variables representing these terms. If there were no incremental terms in the parameters, the terms will be the unchanged parameters. The returned terms are placed as parameters in the term that have an incremental top operator. A variable is introduce for the incremental term and a condition is created with on the left-hand, the introdued variabel combined with the variable S representing the storage and on the right-hand, the incremental term with its parameters substituted.

$$\frac{\begin{array}{r@{\;}l} \mathsf{true} &= \mathsf{isInc}(T_3),\\ \mathit{Conds} \;/\; \mathit{Terms} \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 &= \mathsf{trans\text{-}pars}(T_4',\ \mathit{VarTab},\ \mathit{Svar}),\\ \mathit{Var} \;/\; \mathit{VarTab}_2 &= \mathsf{new\text{-}var}(\mathsf{sort\text{-}of}(T_2),\ \mathit{VarTab}_1),\\ T_5 &= [[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ \mathsf{merge}(T_4,\ \mathit{Terms})],\\ \mathit{Cond} &= \mathsf{new\text{-}cond}(\mathit{Var},\ \mathit{Svar}_1,\ T_5),\\ \mathit{Svar}_2 &= \mathsf{next\text{-}var\text{-}index}(\mathit{Svar}_1) \end{array}}{\begin{array}{l} \mathsf{trans\text{-}pars}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4;\ T_4'],\ \mathit{VarTab},\ \mathit{Svar}) =\\ \mathsf{merge}(\mathit{Conds},\ \mathit{Cond}) \;/\; \mathit{Var} \;/\; \mathit{VarTab}_2 \;/\; \mathit{Svar}_2 \end{array}} \quad \text{[TPars2]}$$

When a parameter does not hav an incremental top operator, its parameters are transformed.

$$\frac{\begin{array}{r@{\;}l} \mathsf{false} &= \mathsf{isInc}(T_3),\\ \mathit{Conds} \;/\; \mathit{Terms} \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 &= \mathsf{trans\text{-}pars}(T_4',\ \mathit{VarTab},\ \mathit{Svar}),\\ T_5 &= [[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ \mathsf{merge}(T_4,\ \mathit{Terms})] \end{array}}{\begin{array}{l} \mathsf{trans\text{-}pars}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4;\ T_4'],\ \mathit{VarTab},\ \mathit{Svar}) =\\ \mathit{Conds} \;/\; T_5 \;/\; \mathit{VarTab}_1 \;/\; \mathit{Svar}_1 \end{array}} \quad \text{[TPars3]}$$

Equation [TPars4] defines the case when the to transform parameter is a list of parameters.

$$\frac{\begin{array}{l} Conds_1 \;/\; Terms_1 \;/\; VarTab_1 \;/\; Svar_1 = \text{trans-pars}(T_1,\ VarTab,\ Svar), \\ Conds_2 \;/\; Terms_2 \;/\; VarTab_2 \;/\; Svar_2 = \text{trans-pars}(T_2,\ VarTab_1,\ Svar_1) \end{array}}{\begin{array}{l} \text{trans-pars}(T_1;\ T_2,\ VarTab,\ Svar) = \\ \text{merge}(Conds_1,\ Conds_2) \;/\; \text{merge}(Terms_1,\ Terms_2) \;/\; VarTab_2 \;/\; Svar_2 \end{array}} \quad \text{[TPars4]}$$

$$\begin{array}{l} \text{trans-pars}(T,\ VarTab,\ Svar) = \\ \text{nil} \;/\; T \;/\; VarTab \;/\; Svar \quad \textbf{otherwise} \end{array} \quad \text{[TPars]}$$

The difference between the function trans-pars defined above and the function trans-terms defined below is that trans-pars introduces a new condition when the input terms has an incremental top operator. The function trans-terms does not introduce new conditions. If the input term has parameters, these parameters will be applied to the function trans-pars.

$$\frac{\begin{array}{r} \text{true} = \text{isInc}(T_3), \\ Conds \;/\; Pars \;/\; VarTab_1 \;/\; Svar_1 = \text{trans-pars}(T_4{}',\ VarTab,\ Svar), \\ T_5 = [[\text{scCfFunction } T_1;\ T_2;\ T_3]\ \text{merge}(T_4,\ Pars)] \end{array}}{\begin{array}{l} \text{trans-terms}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4;\ T_4{}'],\ VarTab,\ Svar) = \\ Conds \;/\; T_5 \;/\; VarTab_1 \;/\; Svar_1 \end{array}} \quad \text{[TTerms1]}$$

$$\frac{\begin{array}{r} \text{false} = \text{isInc}(T_3), \\ Conds \;/\; Pars \;/\; VarTab_1 \;/\; Svar_1 = \text{trans-pars}(T_4{}',\ VarTab,\ Svar), \\ T_5 = [[\text{scCfFunction } T_1;\ T_2;\ T_3]\ \text{merge}(T_4,\ Pars)] \end{array}}{\begin{array}{l} \text{trans-terms}([[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4;\ T_4{}'],\ VarTab,\ Svar) = \\ Conds \;/\; T_5 \;/\; VarTab_1 \;/\; Svar_1 \end{array}} \quad \text{[TTerms2]}$$

$$\frac{\begin{array}{l} Conds_1 \;/\; Terms_1 \;/\; VarTab_1 \;/\; Svar_1 = \text{trans-terms}(T_1,\ VarTab,\ Svar), \\ Conds_2 \;/\; Terms_2 \;/\; VarTab_2 \;/\; Svar_2 = \text{trans-terms}(T_2,\ VarTab_1,\ Svar_1) \end{array}}{\begin{array}{l} \text{trans-terms}(T_1;\ T_2,\ VarTab,\ Svar) = \\ \text{merge}(Conds_1,\ Conds_2) \;/\; \text{merge}(Terms_1,\ Terms_2) \;/\; VarTab_2 \;/\; Svar_2 \end{array}} \quad \text{[TTerms3]}$$

$$\begin{array}{l} \text{trans-terms}(T,\ VarTab,\ Svar) = \\ \text{nil} \;/\; T \;/\; VarTab \;/\; Svar \quad \textbf{otherwise} \end{array} \quad \text{[TTerms]}$$

The function mk-new-lhs-eq defined below returns an ATerm representing the new left-hand side in an unsafe incremental equation. The attribute incremental is removed by means of the function rm-inc-att and the incremental information and the storage are added as arguments.

$$\frac{\begin{array}{l} T_1{}' = \text{addINCINFO-STORAGE}(T_1), \\ T_2{}' = [\text{scSort scNF-STORAGE}], \\ T_3{}' = \text{rm-inc-att}(T_3), \\ T_4{}' = \text{merge}(T_4,\ \text{rank-level-R-L ( ); varS ( )}) \end{array}}{\begin{array}{l} \text{mk-new-lhs-eq}([[\text{scCfFunction } T_1;\ T_2;\ [L\ T_3]]\ T_4]) = \\ [[\text{scCfFunction } T_1{}';\ T_2{}';\ [L\ T_3{}']]\ T_4{}'] \end{array}} \quad \text{[NlhsE]}$$

The function new-rhs defined below returns a new right-hand side of an unsafe equation.

$$\begin{array}{l} \text{new-rhs}(T,\ Term,\ Svar) = \\ [\text{mk-store-att(scstore-fun-att)} \\ \ \text{lex-term}(T,\ \text{scATT-NAME}); \\ \ [\text{sort2nf(sort-of}(Term))\ \ Term]; \\ \ \text{varS}(Svar)] \end{array} \quad \text{[Nrhs]}$$

The function mk-store-par-att-cond returns an ATerm representing a condition in which the function store-par-att is invoked to store the value of a parameter in an attribute in the storage.

mk-store-par-att-cond(T, T_1, T_2) = [MSPA]
[scCondition scEq;
[storage-decl ( ) bool-constant(sctrue); varS(next-var-index(T_2))];
[mk-store-att(scstore-par-att)
lex-term(T, scATT-NAME);
[sort2nf(sort-of(T_1)) T_1];
varS(T_2)]]

The function mk-store-att yields an ATerm representing the ASF+SDF declaration store-par-att(ATT-NAME, NORMAL-FORM, STORAGE) or store-fun-att(ATT-NAME, NORMAL-FORM, STORAGE) where *AN* is the name of the attribute in which the the normal form | *term* | should be stored.

mk-store-att(T) = [MSA]
[scCfFunction
[scCfElems
T;
"("; [scSort scATT-NAME]; ","; [scSort scNORMAL-FORM]; ",";
[scSort scSTORAGE]; ")"];
[scSort scNF-STORAGE];
[scAttributes nil]]

The function *new-cond* returns an ATerm representing an condition with on its left-hand variables introduced to substitute terms with incremental top operators and combined with the storage.

$$\frac{\begin{array}{l} T' = \text{lhs-inc-cond}(T, \mathit{Svar}), \\ T_1' = \text{addSTORAGE}(T_1), \\ T_2' = [\text{scSort scNF-STORAGE}], \\ T_4' = \text{merge}(T_4, \text{varS}(\mathit{Svar})) \end{array}}{\begin{array}{l} \text{new-cond}(T, \mathit{Svar}, [[\text{scCfFunction } T_1;\ T_2;\ T_3]\ T_4]) = \\ [\text{scCondition scEq};\ T';\ [[\text{scCfFunction } T_1';\ T_2';\ T_3]\ T_4']] \end{array}} \quad \text{[NewC]}$$

The function lhs-inc-cond returns an ATerm representing the left-hand side of a condition in in which the combination old left-hand side and storage is created.

lhs-inc-cond(T, *Svar*) = [IClhs]
[storage-decl ( )
[sort2nf(sort-of(T)) T];
varS(next-var-index(*Svar*))]

G.3.7 StatusConditionASDF

In the module StatusConditionASDF the specification is given for the generation of status-is-unsafe or status-is-safe condition.

module StatusConditionASDF
imports AuxMod2IncModASDF$^{G.3.8}$
exports
context-free syntax
mk-status-cond(ATerm, ATerm) → ATerm
hiddens
context-free syntax

mk-status-fun(ATerm, ATerm, ATerm) → ATerm

equations

The function mk-status-cond defined below returns an ATerm representing the condition true:S_1 = status-is-safe(AN,,R:L, S) or true:S_1 = status-is-unsafe(AN,R:L,S) depending on the string constants scSafe and scUnSafe. AN represents the name of the attributes which status is to be checked.

mk-status-cond(T, T_1) = [SCond1]
[scCondition scEq;
[storage-decl ( ) bool-constant(sctrue); varS("1")];
mk-status-fun(T, T_1, rank-level-R-L ( ); varS ( ))]

The function mk-status-fun yields an ATerm representing the ASF+SDF term status-is-safe or status-is-unsafe with arguments.

mk-status-fun(T, T_1, T_2) = [MSF]
[[scCfFunction
[scCfElems
T_1;
"("; [scSort scATT-NAME]; ","; [scSort scINCINFO]; ",";
[scSort scSTORAGE]; ")"];
[scSort scBOOL-STORAGE];
[scAttributes nil]]
lex-term(T, scATT-NAME);
T_2]

G.3.8 AuxMod2IncModASDF

In the module AuxMod2IncModASDF some auxiliary functions have been defined.

module AuxMod2IncModASDF
imports AuxIncModule$^{G.1.3}$
exports
context-free syntax

addSTORAGE(ATerm) → ATerm
addINCINFO-STORAGE(ATerm) → ATerm
varS() → ATerm
varS(ATerm) → ATerm
storage-decl() → ATerm

scNF-STORAGE → Literal

equations

The function addSTORAGE adds the sort STORAGE to the argument list of a function declaration.

addSTORAGE(T) = add2cfelems(T, [scSort scSTORAGE]) [AST]

The function addINCINFO-STORAGE adds the sorts INCINFO and STORAGE to the argument list of a function declaration.

addINCINFO-STORAGE(T) = [AIIS]
add2cfelems(T, [scSort scINCINFO]; ","; [scSort scSTORAGE])

The function varS called without argument returns an ATerm representing the variable S. With argument the same variable is returned by will be given an index.

varS ( ) = var-term("S", scSTORAGE) [varS1]
varS(T) = var-term("S", T, scSTORAGE) [varS2]

The function storage-decl returns an ATerm representing the function declaration: BOOL : STORAGE $\rightarrow$ BOOL-STORAGE.

```
storage-decl ( ) =                                          [Sdecl]
 [scCfFunction
  [scCfElems
   [scSort scBOOL]; [scQLit ":"]; [scSort scSTORAGE]];
  [scSort scBOOL-STORAGE];
  [scAttributes nil]]
```

```
aterms(scNF-STORAGE) = "NF-STORAGE"                         [AT1]
```

G.4 Derive InitUpdate module, external version

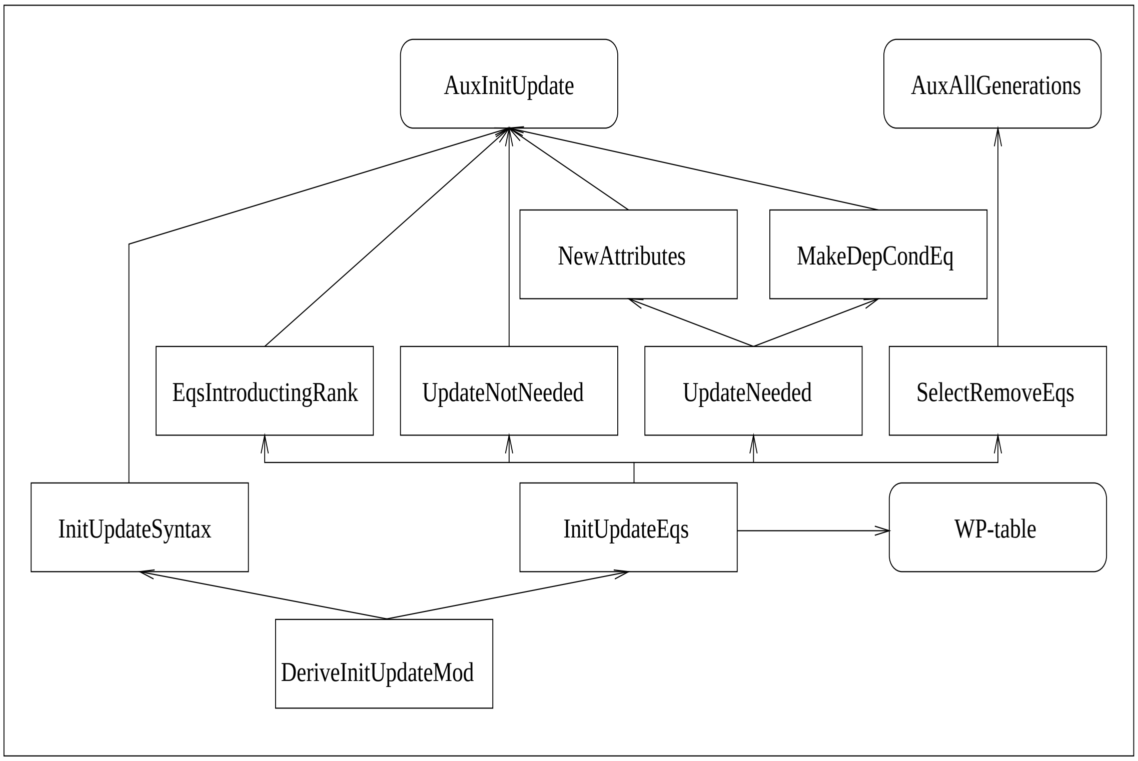


Figure G.4: Module import overview.

G.4.1 DeriveInitUpdateMod

In the module DeriveInitUpdateMod the derivation of an InitUpdate module has been specified.

module DeriveInitUpdateMod
imports InitUpdateSyntax$^{G.4.2}$ InitUpdateEqs$^{G.4.3}$
exports
  context-free syntax
    derive-init-update-mod({ATerm ","}∗) → ATerm
hiddens
  context-free syntax
    module-names({ATerm ","}+) → ATerm
    rm-annotations(ATerm) → ATerm
equations

The function derive-init-update-mod derives an InitUpdate module given a number of modules. First the names of the input modules are selected. The functions that have been declared as incremental in the syntax sections of these modules are selected by means of the function inc-operators-in-syntax. This is only an intermediate result. In the selected function declarations, the sorts of the constructors are selected. All equations in the input modules defining incremental functions are selected by means of the function inc-equations. The result yielded by this functions is annotated with module names. These annotations are not needed and are removed by applying the function rm-annotations. The derived module is given the name `InitUpdate` and its full module name (normally module name and directory) is set to `""`, the empty string. We do not know in which directory the module will be stored. The source syntax and source equations are given value nil. The syntax section is derived by application of init-update-syntax and the equations are created by invoking the function init-update-equations.

$$\frac{\begin{array}{l} T_5 = \text{module-names}(Ts^+),\\ T_6 = \text{inc-operators-in-syntax}(Ts^+),\\ T_7 = \text{constructor-sorts}(T_6),\\ T_8 = \text{rm-annotations}(\text{inc-equations}(Ts^+)),\\ L_1 = \texttt{"InitUpdate"},\\ L_2 = \texttt{""},\\ T_1 = [\text{scSourceSyntax nil}],\\ T_2 = \text{init-update-syntax}(T_5,\ T_7),\\ T_3 = [\text{scSourceEquations nil}],\\ T_4 = \text{init-update-equations}(T_8,\ T_7) \end{array}}{\begin{array}{l} \text{derive-init-update-mod}(Ts^+) =\\ [\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4] \end{array}} \quad \text{[DIUM1]}$$

$$\text{derive-init-update-mod}(Ts) \ = \ \text{nil} \quad \textbf{otherwise} \quad \text{[DIUM]}$$

The function rm-annotations defined below removes the annotations of terms in the input list.

$$\text{rm-annotations}(T_1;\ T_2) = \quad \text{[RA1]}$$
$$\text{merge}(\text{rm-annotations}(T_1), \text{rm-annotations}(T_2))$$

$$\text{rm-annotations}(T_1\ /\ T_2) \ = \ T_1 \quad \text{[RA2]}$$
$$\text{rm-annotations}(T) \ = \ T \quad \textbf{otherwise} \quad \text{[RA]}$$

The function module-names defined below returns the names of the modules in a list (of modules).

$$\text{module-names}(T,\ Ts^+) = \quad \text{[MN1]}$$
$$\text{merge}(\text{module-names}(T), \text{module-names}(Ts^+))$$

$$\text{module-names}([\text{scModule } L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) \ = \ L_1 \quad \text{[MN2]}$$

G.4.2 InitUpdateSyntax

In the module InitUpdateSyntax the function init-update-syntax has been defined. When this function is applied, the syntax section of an InitUpdate module is created.

module InitUpdateSyntax
imports AuxInitUpdate$^{G.4.10}$
exports
  context-free syntax
    init-update-syntax(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    inc-mod-names(ATerm) → ATerm
equations

Below the function init-update-syntax is defined. This function creates the syntax section of the InitUpdate module. Input: T_1 a list containing the sorts of the constructors in incremental functions in the modules for which the InitUpdate module is derived and T_2 is a list containing the names of these modules.

$$\frac{\begin{array}{l} T_3 = \text{init-operators}(T_2), \\ T_4 = \text{init-operators-with-rank}(T_2), \\ T_5 = \text{mk-variables}(T_2), \\ T_6 = \text{inc-mod-names}(T_1) \end{array}}{\begin{array}{l} \text{init-update-syntax}(T_1,\ T_2) = \\ [\text{scPrefixSyntax} \\ \ [\text{scImports} \\ \ \ [\text{scId}\ \texttt{"StorageInterfaceInit"}]; \\ \ \ T_6]; \\ \ [\text{scExports} \\ \ \ [\text{scContextFreeSyntax}\ T_3]]; \\ \ [\text{scHiddens} \\ \ \ [\text{scContextFreeSyntax}\ T_4]; \\ \ \ [\text{scVariables}\ T_5]]] \end{array}} \quad \text{[IUS]}$$

The function inc-mod-names defined below puts the prefix `Inc` before each module name.

inc-mod-names(T_1; T_2) = [IMN1]
merge(inc-mod-names(T_1), inc-mod-names(T_2))

inc-mod-names(L) = [IMN2]
[scId cc("`Inc`"; L)]

G.4.3 InitUpdateEqs

In the module InitUpdateEqs the generation of the equations for an InitUpdate has been specified.

module InitUpdateEqs
imports SelectRemoveEqs$^{G.4.4}$ EqsIntroducingRank$^{G.4.5}$ WP-table$^{E.5.1}$
imports UpdateNeededEq$^{G.4.7}$ UpdateNotNeededEq$^{G.4.6}$
exports
  context-free syntax
    init-update-equations(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    init-update-eqs(ATerm, ATerm, ATerm, INT) → ATerm
    mk-eqs(ATerm, ATerm, INT) → ATerm
    mk-lhs-eq(ATerm) → ATerm
    simplify-wp-table(ATerm) → ATerm
    rank-children(ATerm, ATerm) → ATerm
    rank-children(ATerm, INT, ATerm) → ATerm
    const-inc-fun(ATerm, ATerm) → BOOL
    inj-equations(ATerm, INT) → ATerm
    mk-inj-eq(ATerm, INT) → ATerm
equations

When the function init-update-equations defined below is applied, the equations for the InitUpdate module are created. Input: T are all equations with incremental top operators on their left-hand sides and T' are the sorts of the constructors. The equations introducing the rank are generated by applying the function eqs-intro-rank to the constructor sorts. The incremental equations are restructed by means of the function restruct-eqs. This facilitates the specification of the generation of the rest of the equations. Two sets of equations are made: T_3 containing all equations not defining injections and T_4 containing the equations defining injections. When the parameter attributes are created at a later stage, their definition rules are needed. A simplified version of the wp-table is created. The "1" given as argument when the functions eqs-intro-rank and init-update-eqs are called will be used to number the tags of the generated equations.

$$\frac{\begin{array}{l} T_1 = \text{eqs-intro-rank}(T', 1), \\ T_2 = \text{restruct-eqs}(T), \\ T_3 = \text{rm-injection-eqs}(T_2), \\ T_4 = \text{select-injection-eqs}(T_2), \\ T_5 = \text{simplify-wp-table}(\text{wp-table}(T)), \\ T_6 = \text{init-update-eqs}(T_3,\ T_4,\ T_5,\ 1) \end{array}}{\begin{array}{l} \text{init-update-equations}(T,\ T') = \\ [\text{scPrefixEquations merge}(T_1,\ T_6)] \end{array}} \qquad \text{[IUEQs1]}$$

The function init-update-eqs is defined below. Equation [IUEqs2] applies when there is only one equation for which a new equation should be created. In equation [IUEqs2], (input is a list of equations) all equations defining an incremental function with the same constructor are selected and are together with equations defining injections for the same sort applied to the function mk-eqs. The simplified table with parameter attribute definitions, T_3 and the integer *int* (the tag index) are also passed to this function. After the equations defining functions for the constructor have been remove, the function init-update-eqs is called recursively.

$$\frac{\begin{array}{l} T_4 = \text{eqs-same-constructor}(T,\ T_1), \\ T_5 = \text{inj-eqs-same-constructor}(T,\ T_2), \\ T_6 = \text{mk-eqs}(\text{merge}(T;\ T_4,\ T_5),\ T_3,\ \mathit{Int}), \\ T_7 = \text{rm-eqs-same-constructor}(T,\ T_1), \\ T_8 = \text{init-update-eqs}(T_7,\ T_2,\ T_3,\ \mathit{Int}+1) \end{array}}{\text{init-update-eqs}(T;\ T_1,\ T_2,\ T_3,\ \mathit{Int}) = \text{merge}(T_6,\ T_8)} \qquad \text{[IUEqs1]}$$

$$\frac{\begin{array}{l} T_5 = \text{inj-eqs-same-constructor}([T\ T_1],\ T_2), \\ T_6 = \text{mk-eqs}(\text{merge}([T\ T_1],\ T_5),\ T_3,\ \mathit{Int}), \\ T_7 = \text{inj-equations}(T_2,\ 1) \end{array}}{\begin{array}{l} \text{init-update-eqs}([T\ T_1],\ T_2,\ T_3,\ \mathit{Int}) = \\ \text{merge}(T_6,\ T_7) \end{array}} \qquad \text{[IUEqs2]}$$

$$\begin{array}{l} \text{init-update-eqs}(T_1,\ T_2,\ T_3,\ \mathit{Int}) = \\ \text{inj-equations}(T_2,\ 1) \quad \textbf{otherwise} \end{array} \qquad \text{[IUEqs]}$$

The function mk-eqs defined below creates two equations defining the function init in the InitUpdate module. T_1 contains a list of equations defined for the same constructor. The left-hand side of the two generated equations will be the same thus the left-hand side is generated by means of the function mk-lhs-eq and can be passed as an argument to the functions update-not-needed-eq and update-needed-eq. The list of ranks of the children will also be passed as an argument to these two functions and will be supplied by applying the function rank-children.

$$\frac{\begin{array}{l} T_3 = \text{mk-lhs-eq}(T_1), \\ T_6 = \text{rank-children}(T_3,\ T_1), \\ T_4 = \text{update-not-needed-eq}(T_3,\ T_6,\ \mathit{Int}), \\ T_5 = \text{update-needed-eq}(T_3,\ T_1,\ T_2,\ T_6,\ \mathit{Int}) \end{array}}{\text{mk-eqs}(T_1,\ T_2,\ \mathit{Int}) = \text{merge}(T_4,\ T_5)} \qquad \text{[MEqs1]}$$

The function mk-lhs-eq defined below creates a new left-hand side for a list of equations defined for the same constructor.

$$\text{mk-lhs-eq}(T_1;\ T_2) \ = \ \text{mk-lhs-eq}(T_1) \qquad \text{[LHS1]}$$

$$\text{mk-lhs-eq}([\text{scCondEquation } L_1;\ L_2;\ [T_1\ \ T_2;\ T_3];\ T_4;\ T_5]) = \qquad \text{[LHS2]}$$
$$\text{mk-lhs-eq}([\text{scCondEquation } L_1;\ L_2;\ [T_1\ \ T_2];\ T_4;\ T_5])$$

$$\frac{\begin{array}{l} T_6 = \text{init-operators-with-rank}([\text{scSort first-sort}(T_1)]), \\ T_7 = \text{varR ( )} \end{array}}{\begin{array}{l} \text{mk-lhs-eq}([\text{scCondEquation } L_1;\ L_2;\ [T_1\ [T_2\ \ T_3]];\ T_4;\ T_5]) = \\ [T_6\ [T_2\ \ T_3];\ T_7] \end{array}} \qquad \text{[LHS3]}$$

When the names of the parameter attributes are generated, we need to know if some of them are uniquely defined. The function simplify-wp-table simplifies the "real" wp-table. Only the names of the attributes are needed and the rest of the information in an entry in the wp-table is removed.

$$\text{simplify-wp-table}(T_1;\ T_2) = \qquad \text{[SWP1]}$$
$$\text{merge}(\text{simplify-wp-table}(T_1), \text{simplify-wp-table}(T_2))$$

$$\text{simplify-wp-table}([[T_1\ \ T]\ T_2]\ /\ T') \ = \ [T_1\ \ T_2] \qquad \text{[SWP2]}$$

$$\text{simplify-wp-table}(T) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[SWP]}$$

When the function rank-children defined below is applied to a list of ATerms representing equations, a list is returned containing the ranks of the children. For all children (subterms) of the constructor on the left-hand side occurring as first argument in a term with an incremental top operator, its child number is selected. If the child is not a constructor in an incremental subterm, the rank 0 will be given to the child of the list. 1;0;3 : The first and third child are constructors in incremental terms, the second is not.

$$\frac{\begin{array}{l} [T_4\ \ T_5] = \text{constructor}(T_1), \\ T_6 = \text{inc-terms}(T_3) \end{array}}{\begin{array}{l} \text{rank-children}(T_1,\ T_3) = \\ \text{rank-children}(T_5,\ 1,\ T_6) \end{array}} \qquad \text{[RankC1]}$$

$$\text{rank-children}(T_1;\ T_2,\ Int,\ T_6) = \qquad \text{[RankC2]}$$
$$\text{merge}(\text{rank-children}(T_1,\ Int,\ T_6), \text{rank-children}(T_2,\ Int + 1,\ T_6))$$

$$\text{rank-children}([\text{scVar } T],\ Int,\ T_1) = \text{int2lit}(Int) \qquad \text{[RankC3]}$$
$$\textbf{when}$$
$$\text{true} = \text{const-inc-fun}([\text{scVar } T]\ /\ \text{int2lit}(Int),\ T_1)$$

$$\text{rank-children}(T,\ Int,\ T_1) \ = \ \texttt{"0"} \quad \textbf{otherwise} \qquad \text{[RankC]}$$

The function const-inc-fun is a boolean function yielding true when T, a variable annotated with a child number, is present in the equations, the function's second argument.

$$\text{const-inc-fun}(T,\ T_1;\ T_2) \ = \ \text{const-inc-fun}(T,\ T_1) \mid \text{const-inc-fun}(T,\ T_2) \qquad \text{[CIF1]}$$
$$\text{const-inc-fun}(T,\ [T_1\ \ T_2;\ T_3]) \ = \ \text{occurs}(T,\ T_2) \qquad \text{[CIF2]}$$

$$\text{const-inc-fun}(T,\ [T_1\ \ T_2]) \ = \ \text{occurs}(T,\ T_2) \quad \textbf{otherwise} \qquad \text{[CIF]}$$

Below the function inj-equations is defined. Default equations are created for equations defining injection for the same constructor.

$$\frac{\begin{array}{l} T_4 = \text{mk-inj-eq}(T_1,\ \mathit{Int}), \\ T_5 = \text{rm-eqs-same-constructor}(T_1,\ T_2), \\ T_6 = \text{inj-equations}(T_5,\ \mathit{Int} + 1) \end{array}}{\text{inj-equations}(T_1;\ T_2,\ \mathit{Int}) = \text{merge}(T_4,\ T_6)} \quad \text{[InjEq1]}$$

$$\text{inj-equations}(T,\ \mathit{Int}) = \text{mk-inj-eq}(T,\ \mathit{Int}) \quad \text{[InjEq2]}$$

$$\text{inj-equations}(T,\ \mathit{Int}) = \text{nil} \quad \textbf{otherwise} \quad \text{[InjEq]}$$

The actual creation of the equations representing equations defining injection is defined below.

$$\frac{\begin{array}{r} [T_6\ T_6'] = \text{constructor}(T_1), \\ T_4\ /\ T_5 = \text{injections}([T_6\ T_6']), \\ T_7 = \text{init-operators-with-rank}(T_5), \\ T_8 = \text{init-operators-with-rank}(T_4), \\ T_9 = \text{sort2var}(T_5), \\ T_{10} = \text{sort2var}(T_4) \end{array}}{\begin{array}{l} \text{mk-inj-eq}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3],\ \mathit{Int}) = \\ [\text{scCondEquation} \\ \ \text{scImplies}; \\ \ \text{tag}(\texttt{"default-INJ"},\ \mathit{Int}); \\ \ [T_7\ [T_6\ T_9];\ \text{varR}\ (\ )]; \\ \ [T_8\ T_{10};\ \text{varR}\ (\ )]; \\ \ [\text{scConditions} \\ \ \ [\text{scCondition scEq};\ T_9;\ T_{10}]]] \end{array}} \quad \text{[MIEq]}$$

$$\text{mk-inj-eq}(T,\ \mathit{Int}) = \text{nil} \quad \textbf{otherwise} \quad \text{[MIEq]}$$

G.4.4 SelectRemoveEqs

In the module SelectRemoveEqs functions have been defined to select and remove equations defined for certain constructors or defining an injection.

module SelectRemoveEqs
imports AuxAllGenerations$^{G.1.2}$
exports
  context-free syntax
    select-injection-eqs(ATerm) → ATerm
    rm-injection-eqs(ATerm) → ATerm
    eqs-same-constructor(ATerm, ATerm) → ATerm
    inj-eqs-same-constructor(ATerm, ATerm) → ATerm
    rm-eqs-same-constructor(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    is-injection-eq(ATerm) → BOOL
    is-inj(ATerm) → BOOL
equations

The function select-injection-eqs defined below yields a list containing all equations defining injections in the input, a list of equations.

$$\text{select-injection-eqs}(T_1;\ T_2) = \qquad \text{[SIEqs1]}$$
$$\text{merge}(\text{select-injection-eqs}(T_1),\ \text{select-injection-eqs}(T_2))$$

$$\text{select-injection-eqs}([\text{scCondEquation } T]) = [\text{scCondEquation } T] \qquad \text{[SIEqs2]}$$
$$\textbf{when}$$
$$\text{true} = \text{is-injection-eq}([\text{scCondEquation } T])$$

$$\text{select-injection-eqs}(T) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[SIEqs]}$$

The function rm-injection-eqs defined below yields a list of equations that do not define injections, thus the equations defining injections are removed in the input: a list of equations.

$$\text{rm-injection-eqs}(T_1;\ T_2) = \qquad \text{[RMIEs1]}$$
$$\text{merge}(\text{rm-injection-eqs}(T_1),\ \text{rm-injection-eqs}(T_2))$$

$$\text{rm-injection-eqs}([\text{scCondEquation } T]) = [\text{scCondEquation } T] \qquad \text{[RMIEs2]}$$
$$\textbf{when}$$
$$\text{false} = \text{is-injection-eq}([\text{scCondEquation } T])$$

$$\text{rm-injection-eqs}(T) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[RMIEs]}$$

The function eqs-same-constructor defined below yields a list of equations that are defined for the same constructor as its first argument, an equation. The second argument is a list of equations.

$$\text{eqs-same-constructor}(T,\ T_1;\ T_2) = \qquad \text{[ESC1]}$$
$$\text{merge}(\text{eqs-same-constructor}(T,\ T_1),\ \text{eqs-same-constructor}(T,\ T_2))$$

$$\frac{\begin{array}{c}\text{constructor}(T_1) = [T\ T_4],\\ \text{constructor}(T_1{}') = [T'\ T_5],\\ T = T'\end{array}}{\begin{array}{l}\text{eqs-same-constructor}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\\ \qquad [\text{scCondEquation } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}']) =\\ [\text{scCondEquation } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}']\end{array}} \qquad \text{[ESC2]}$$

$$\text{eqs-same-constructor}(T,\ T_1) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[ESC]}$$

The function inj-eqs-same-constructor selects all equations defining injections that are defined for the same constructor as the equation in the first argument of the function. The second argument is a list of equations defining injections.

$$\text{inj-eqs-same-constructor}(T,\ T_1;\ T_2) = \qquad \text{[IESC1]}$$
$$\text{merge}(\text{inj-eqs-same-constructor}(T,\ T_1),\ \text{inj-eqs-same-constructor}(T,\ T_2))$$

$$\frac{\begin{array}{c}[T_4\ \ T_5] = \text{constructor}(T_1{}'),\\ \text{first-sort}(\text{operator}(T_1)) = \text{first-sort}(T_4)\end{array}}{\begin{array}{l}\text{inj-eqs-same-constructor}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\\ \qquad [\text{scCondEquation } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}']) =\\ [\text{scCondEquation } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}']\end{array}} \qquad \text{[IESC2]}$$

$$\text{inj-eqs-same-constructor}(T,\ T_1) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[IESC]}$$

The function rm-eqs-same-constructor defined below yields its second argument, a list of equations after that all equations defined for the same constructor as its first argument have been removed.

$$\text{rm-eqs-same-constructor}(T,\ T_1;\ T_2) = \qquad \text{[RESC1]}$$
$$\text{merge}(\text{rm-eqs-same-constructor}(T,\ T_1),\ \text{rm-eqs-same-constructor}(T,\ T_2))$$

$$\frac{\begin{array}{c}[T_4\ T_5] = \text{constructor}(T_1),\\ [T_6\ T_7] = \text{constructor}(T_1{}'),\\ T_4 = T_6\end{array}}{\begin{array}{l}\text{rm-eqs-same-constructor}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3],\\ \qquad [\text{scCondEquation } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}']) = \text{nil}\end{array}} \qquad \text{[RESC2]}$$

$$\text{rm-eqs-same-constructor}(T,\ T_1) \ = \ T_1 \quad \textbf{otherwise} \qquad \text{[RESC]}$$

The function is-injection-eq defined below is a boolean function that determines whether the input equation defines an injection or not.

$$\frac{T_4 = \text{inc-terms}(T_2;\ T_3)}{\text{is-injection-eq}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{is-inj}(T_4)} \qquad \text{[IIE1]}$$

The function is-inj defined below returns true when all the terms in the input have been annotated with rank 0 (an injection rank) otherwise the function yields false.

$$\text{is-inj}(T_1;\ T_2) = \text{is-inj}(T_1)\ \&\ \text{is-inj}(T_2) \qquad \text{[IInj1]}$$
$$\text{is-inj}([[\text{scCfFunction } T]\ T_1\ /\ \text{"0"};\ T_2]) = \text{true} \qquad \text{[IInj2]}$$
$$\text{is-inj}([[\text{scCfFunction } T]\ T_1\ /\ \text{"0"}]) = \text{true} \qquad \text{[IInj3]}$$
$$\text{is-inj}(T) = \text{false} \qquad \textbf{otherwise} \qquad \text{[IInj]}$$

G.4.5 EqsIntroducingRank

In the module EqsIntroducingRank the generation of equations in which the rank 1 is added to the term that is being reduced has been specified.

module EqsIntroducingRank
imports AuxInitUpdate$^{G.4.10}$
exports
  context-free syntax
    eqs-intro-rank(ATerm, INT) $\rightarrow$ ATerm
equations
The function eqs-intro-rank defined below creates equations in which the rank 1 is added to the term that is being reduced. Input: a list of sorts for which these equations must be defined. Output: init(*SORTvar*) = init(*SORTvar,1*). The integer *Int* is used to number the equation tags.

$$\text{eqs-intro-rank}(T_1;\ T_2,\ Int) = \qquad \text{[EIR1]}$$
$$\text{merge}(\text{eqs-intro-rank}(T_1,\ Int),\ \text{eqs-intro-rank}(T_2,\ Int + 1))$$

$$\frac{\begin{array}{l} T_1 = \text{init-operators}([L\ T]), \\ T_2 = \text{init-operators-with-rank}([L\ T]), \\ T_3 = \text{sort2var}(T), \\ T_4 = \text{lex-term}(\texttt{"1"}, \text{scINT}) \end{array}}{\begin{array}{l} \text{eqs-intro-rank}([L\ T], \mathit{Int}) = \\ [\text{scCondEquation} \\ \ \text{scImplies}; \\ \ \text{tag}(\texttt{"IR"}, \mathit{Int}); \\ \ [T_1\ T_3]; \\ \ [T_2\ T_3;\ T_4]; \\ \ [\text{scConditions nil}]] \end{array}} \quad \text{[EIR2]}$$

G.4.6 UpdateNotNeededEq

In the module UpdateNotNeededEq a function update-not-needed-eq is defined. This function returns an ATerm representing an equation for the case when no update is needed of a part of the storage.

module UpdateNotNeededEq
imports AuxInitUpdate$^{G.4.10}$
exports
context-free syntax
update-not-needed-eq(ATerm, ATerm, INT) → ATerm
equations

Input: T_1 is the already created left-hand side. T_4 contains the ranks that should be added in the function call update-needed in the condition of the created equation. L_2 will be the tag of the created equation and the right-hand side is given the value true. The update-needed condition is created by applying the function iu-cond to the literal false and the AsFixTerm representing the iteration of the children, yielded by the function mk-iter.

$$\frac{\begin{array}{l} L_1 = \text{scImplies}, \\ L_2 = \text{cc}(\texttt{"In"};\ \text{int2lit}(\mathit{Int});\ \texttt{"a"}), \\ T_2 = \text{bool-constant}(\text{sctrue}), \\ T_3 = [\text{scConditions iu-cond}(\text{scfalse}, \text{mk-iter}(T_4))] \end{array}}{\begin{array}{l} \text{update-not-needed-eq}(T_1, T_4, \mathit{Int}) = \\ [\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3] \end{array}} \quad \text{[UNNeq1]}$$

G.4.7 UpdateNeededEq

In the module UpdateNeededEq the generation of an equation for the case when update is needed has been specified.

module UpdateNeededEq
imports NewAttributes$^{G.4.8}$ MakeDepCondEq$^{G.4.9}$
exports
context-free syntax
update-needed-eq(ATerm, ATerm, ATerm, ATerm, INT) → ATerm
hiddens
context-free syntax
mk-rhs-i-u-needed(ATerm) → ATerm
mk-conds-i-u-needed(ATerm, ATerm, ATerm, ATerm) → ATerm
init-cond(ATerm, ATerm) → ATerm

new-fun-att-cond(ATerm) → ATerm
new-par-att-cond(ATerm) → ATerm
att-iter(ATerm) → ATerm

equations

The function update-needed-eq defined below creates an equation defining the functioninit when update of the (sub) storage is needed. Input: T_1 is the left-hand side the equation should have, T_2 is a list of equations defining functions for a particular constructor, T_3 contains the definition rules of the parameter attributes and T_4 are the ranks of the children of the constructor. The variable *Int* indicates which index the tag should have. Dependencies are derived for the attributes in the equation and the function and parameter attributes are selected. The right-hand side of the new equation is yielded when the function mk-rhs-i-u-needed is applied. By calling the function mk-conds-i-u-needed the conditions are generated.

$$\frac{\begin{array}{l} L_1 = \mathsf{scImplies}, \\ L_2 = \mathsf{cc}(\texttt{"In"};\ \mathsf{int2lit}(\mathit{Int});\ \texttt{"b"}), \\ T_5 = \mathsf{rmdoubles}(\mathsf{makedep\text{-}condeq}(T_2)), \\ T_6 = \mathsf{get\text{-}fun\text{-}atts}(T_2), \\ T_7 = \mathsf{get\text{-}par\text{-}atts}(T_2,\ T_3), \\ T_2{}' = \mathsf{mk\text{-}rhs\text{-}i\text{-}u\text{-}needed}(T_5), \\ T_3{}' = \mathsf{mk\text{-}conds\text{-}i\text{-}u\text{-}needed}(\mathsf{constructor}(T_1),\ T_4,\ T_6,\ T_7) \end{array}}{\begin{array}{l} \mathsf{update\text{-}needed\text{-}eq}(T_1,\ T_2,\ T_3,\ T_4,\ \mathit{Int}) = \\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2{}';\ [\mathsf{scConditions}\ T_3{}']] \end{array}} \quad \text{[UN1]}$$

The function mk-rhs-i-u-needed defined below returns the right-hand side of an update needed equation given the dependencies the application of the function add-deps should have (the right-hand side of the created equation invokes this function). T is a list of ATerms representing the dependencies.

$$\frac{T_1 = \mathsf{iter\text{-}decl}(\mathsf{scDEP},\ \texttt{"."},\ \texttt{"*"})}{\begin{array}{l} \mathsf{mk\text{-}rhs\text{-}i\text{-}u\text{-}needed}(T) = \\ [[\mathsf{scCfFunction} \\ \quad [\mathsf{scCfElems\ scadd\text{-}deps};\ \texttt{"("};\ T_1;\ \texttt{")"}]; \\ \quad [\mathsf{scSort\ scDEPS}]; \\ \quad [\mathsf{scAttributes\ nil}]] \\ \ [T_1\ T]] \end{array}} \quad \text{[RhsEq]}$$

Equation [CUN1] defines the function mk-conds-i-u-needed. An update needed conditions is generated by calling the function iu-cond with the constant true and the ranks T_1. New conditions for adding new function and parameter attributes to the storage are created by means of the functions new-fun-att-cond and new-par-att-cond. Conditions in which subterms are applied to the function init are generated by invoking the function init-cond with the constructor for which the equation is generated and the ranks of its children, T and T_1.

$$\frac{\begin{array}{l} T_4 = \mathsf{iu\text{-}cond}(\mathsf{sctrue},\ \mathsf{mk\text{-}iter}(T_1)), \\ T_5 = \mathsf{new\text{-}fun\text{-}att\text{-}cond}(T_2), \\ T_6 = \mathsf{new\text{-}par\text{-}att\text{-}cond}(T_3), \\ T_7 = \mathsf{init\text{-}cond}(T,\ T_1) \end{array}}{\begin{array}{l} \mathsf{mk\text{-}conds\text{-}i\text{-}u\text{-}needed}(T,\ T_1,\ T_2,\ T_3) = \\ \mathsf{merge}(\mathsf{merge}(T_4,\ T_5),\ \mathsf{merge}(T_6,\ T_7)) \end{array}} \quad \text{[CUN1]}$$

Below the function init-cond is defined. Its first argument is a constructor and the second argument are the ranks of its children. When the rank of a child is 0, equation [InitC3], no condition is created.

$$\begin{array}{l} \mathsf{init\text{-}cond}([T_1\ T_2],\ T_3) = \mathsf{init\text{-}cond}(T_2,\ T_3) \\ \qquad\qquad\qquad\qquad \textbf{when} \\ \qquad\qquad\qquad\qquad\quad T_1 \neq \mathsf{scVar} \end{array} \quad \text{[InitC1]}$$

init-cond(T_1; T_2, T_3; T_4) = merge(init-cond(T_1, T_3), init-cond(T_2, T_4)) [InitC2]

init-cond([scVar T], "0") = nil [InitC3]

init-cond([scVar T_1; T_2], T_3) = [InitC]

[scCondition **otherwise**
scEq;
var-term(scBool, scBOOL);
[init-operators-with-rank(T_2) [scVar T_1; T_2]; lex-term(T_3, scINT)]]

Below the definition of the function new-fun-att-cond is given. For each function attribute in the input, a condition is created with on the left-hand side the variable *Bool* and on the right-hand side the function call new-fun-att(*AN*), where *AN* is the name of the function attribute.

new-fun-att-cond(T_1; T_2) = [NFAttC1]
merge(new-fun-att-cond(T_1), new-fun-att-cond(T_2))

new-fun-att-cond(L) = [NFAttC2]
[scCondition
scEq;
var-term(scBool, scBOOL);
[[scCfFunction
[scCfElems scnew-fun-att; "("; [scSort scATT-NAME]; ")"];
[scSort scBOOL];
[scAttributes nil]]
lex-term(L, scATT-NAME)]]

new-fun-att-cond(T) = nil **otherwise** [NFAttC]

The name of a parameter attribute can be a list of attribute names when attributes are equally defined. When the name of an attribute consists of only one name, it is a list of attribute names with one object. For each of these lists a condition is created with on the right-hand side the variable *Bool* and on the left-hand side the function call new-par-att(*ANs*), with *ANs* being the names of the parameter attribute. In equation [NPAttC], T_1 represents a list of attribute names and T_2 is a constant that is used to distinguish the lists of names in the list of attributes.

new-par-att-cond(T_1; T_2) = [NPAttC1]
merge(new-par-att-cond(T_1), new-par-att-cond(T_2))

T_3 = iter-decl(scATT-NAME, ";", "+")
________________________________ [NPAttC2]
new-par-att-cond(T_1 / T_2) =
[scCondition
scEq;
var-term(scBool, scBOOL);
[[scCfFunction
[scCfElems scnew-par-att; "("; T_3; ")"];
[scSort scBOOL];
[scAttributes nil]]
[T_3 att-iter(T_1)]]]

new-par-att-cond(T) = nil **otherwise** [NPAttC]

The function att-iter defined below creates an AsFixTerm representing the attribute name for each name in the input list.

att-iter(T_1; T_2) = merge(att-iter(T_1), att-iter(T_2)) [AttIter1]

att-iter(L) = lex-term(L, scATT-NAME) [AttIter2]

G.4.8 NewAttributes

In the module NewAttributes functions have been specified to collect all function and parameter attributes in a list of ATerms representing equations. When parameter attributes are equally defined, their names are merged together to form the name of one attribute.

module NewAttributes
imports AuxInitUpdate$^{G.4.10}$
exports
  context-free syntax
    get-fun-atts(ATerm) → ATerm
    get-par-atts(ATerm, ATerm) → ATerm
hiddens
  context-free syntax
    get-par-atts-eq(ATerm) → ATerm
    get-par-atts-pars(ATerm, ATerm, INT) → ATerm

    merge-atts-equally-defined(ATerm, ATerm) → ATerm
    select(ATerm, ATerm, ATerm) → ATerm
    remove-atts(ATerm, ATerm) → ATerm
    defined-by(ATerm, ATerm) → ATerm
    defined-by(ATerm, ATerm, ATerm) → ATerm
    att-iter(ATerm) → ATerm
equations

The function get-fun-atts defined below selects the top operator of the left-hand side of each equation in the input and creates the name of the related function attributes by means of the functions mk-decl and fun-att.

$$\text{get-fun-atts}(T_1;\ T_2) = \text{merge}(\text{get-fun-atts}(T_1),\ \text{get-fun-atts}(T_2)) \quad \text{[FAtts1]}$$

$$\text{get-fun-atts}([\text{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \text{fun-att}(\text{mk-decl}(T_1)) \quad \text{[FAtts2]}$$

$$\text{get-fun-atts}(T)\ =\ \text{nil} \quad \textbf{otherwise} \quad \text{[FAtts]}$$

Below the function get-par-atts is defined. First the parameter attributes in the input equations are found by means of the function get-par-atts-eq. Given the result of this application, attributes that are equally defined are merged together when the function merge-atts-equally-defined is applied. T_2 contains the definition rules of the parameter attributes. The information in this table is needed to determine the definitions of the attributes.

$$\frac{T_3 = \text{rmdoubles}(\text{get-par-atts-eq}(T_1)),\quad T_4 = \text{merge-atts-equally-defined}(T_3,\ T_2)}{\text{get-par-atts}(T_1,\ T_2) = T_4} \quad \text{[PAtts]}$$

The function get-par-atts-eq defined below returns all parameter attributes in a list of equations. When the left-hand side of an equation has parameters, T_3 in equation [PAttsE2], parameter attributes are made of these parameters by application of the function get-par-atts-pars. The string representation of the function declaration of the top operator on this left-hand side is needed and provided by application of the function mk-decl. The integer 1 will be used to number the attributes.

$$\text{get-par-atts-eq}(T_1;\ T_2)\ =\ \text{merge}(\text{get-par-atts-eq}(T_1),\ \text{get-par-atts-eq}(T_2)) \quad \text{[PAttsE1]}$$

$$\text{get-par-atts-eq}([\text{scCondEquation } L_1;\ L_2;\ [T_1\ \ T_2;\ T_3];\ T_4;\ T_5]) = \text{get-par-atts-pars}(\text{mk-decl}([T_1\ \ T_2;\ T_3]),\ T_3,\ 1) \qquad \text{[PAttsE2]}$$

$$\text{get-par-atts-eq}(T) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[PAttsE]}$$

For each parameter in the input list, a new parameter attribute name is created. Equation [PAttsP2] applies when the second argument is one parameters. In the equation, this is represented by: $[T_1 \ \ T_2]$.

$$\text{get-par-atts-pars}(T,\ T_1;\ T_2,\ Int) = \text{merge}(\text{get-par-atts-pars}(T,\ T_1,\ Int),\ \text{get-par-atts-pars}(T,\ T_2,\ Int + 1)) \qquad \text{[PAttsP1]}$$

$$\text{get-par-atts-pars}(T,\ [T_1\ \ T_2],\ Int) \ = \ \text{par-att}(T,\ Int) \qquad \text{[PAttsP2]}$$

$$\text{get-par-atts-pars}(T,\ T_1,\ Int) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[PAttsP]}$$

The function merge-atts-equally-defined selects all attributes that are defined in the same way as T_1 in T_2 using the table of definition rules T_3. The attributes in T_2 that were selected are removed by means of the function remove-atts. The list of attributes equally defined is annotated so that we can distinguish the lists from the lists.

$$\frac{\begin{array}{l} T_4 = \text{select}(\text{defined-by}(T_1,\ T_3),\ T_2,\ T_3), \\ T_5 = \text{remove-atts}(T_4,\ T_2), \\ T_6 = \text{merge}(T_1,\ T_4)\ /\ \text{scEqDef}, \\ T_7 = \text{merge-atts-equally-defined}(T_5,\ T_3) \end{array}}{\text{merge-atts-equally-defined}(T_1;\ T_2,\ T_3) = \text{merge}(T_6,\ T_7)} \qquad \text{[MAEDef1]}$$

$$\text{merge-atts-equally-defined}(L,\ T_1) \ = \ L\ /\ \text{scEqDef} \qquad \text{[MAEDef2]}$$

$$\text{merge-atts-equally-defined}(\text{nil},\ T) \ = \ \text{nil} \qquad \text{[MAEDef3]}$$

Below the function select is defined. When this function is applied, all attributes in the second argument defined in the same way as the attribute represented by the varible T are returned.

$$\text{select}(T,\ T_1;\ T_2,\ T_3) = \text{merge}(\text{select}(T,\ T_1,\ T_3),\ \text{select}(T,\ T_2,\ T_3)) \qquad \text{[Sel1]}$$

$$\text{select}(T,\ L,\ T_3) \ = \ L \quad \textbf{when} \ \ T = \text{defined-by}(L,\ T_3) \qquad \text{[Sel2]}$$

$$\text{select}(T_1,\ T_2,\ T_3) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[Sel]}$$

The functionremove-atts removes all objects in its first argument out of its second argument.

$$\text{remove-atts}(T_1;\ T_2,\ T_3) \ = \ \text{merge}(\text{remove-atts}(T_1,\ T_3),\ \text{remove-atts}(T_2,\ T_3)) \qquad \text{[RMAs1]}$$

$$\text{remove-atts}(L,\ T_1;\ T_2) \ = \ \text{merge}(\text{remove-atts}(L,\ T_1),\ \text{remove-atts}(L,\ T_2)) \qquad \text{[RMAs2]}$$

$$\text{remove-atts}(L,\ L) \ = \ \text{nil} \qquad \text{[RMAs3]}$$

$$\text{remove-atts}(T,\ T_1) \ = \ T_1 \quad \textbf{otherwise} \qquad \text{[RMAs]}$$

The function defined-by defined below returns the name of the attribute defining the attribute in its first argument. When no definition is found the first argument is returned as result.

$$\text{defined-by}(T,\ T_1) \ = \ \text{defined-by}(T,\ T_1,\ T_1) \qquad \text{[DefBy1]}$$

$$\text{defined-by}(T,\ [T\ T_1];\ T_2,\ T_3) \ = \ \text{defined-by}(T_1,\ T_2,\ T_2) \qquad \text{[DefBy2]}$$

$$\frac{T \neq T'}{\textsf{defined-by}(T,\ [T'\ T_1];\ T_2,\ T_3) = \textsf{defined-by}(T,\ T_2,\ T_3)} \quad \text{[DefBy3]}$$

$$\textsf{defined-by}(T,\ [T\ T_1],\ T_3) \ = \ \textsf{defined-by}(T_1,\ T_3,\ T_3) \quad \text{[DefBy4]}$$

$$\textsf{defined-by}(T,\ T_1,\ T_2) \ = \ T \qquad \textbf{otherwise} \quad \text{[DefBy]}$$

G.4.9 MakeDepCondEq

In the module MakeDpeCondEq the specification is given for deriving attributes dependencies given a number of equations in AsFix representation. Before the equations are applied to the function makedep-condeq defined in this module they should have been applied to the function restruct-eqs that has been defined in the module Restruct-eqs.

module MakeDpeCondEq
imports AuxInitPath$^{G.1.4}$
exports
  context-free syntax
    makedep-condeq(ATerm) → ATerm
hiddens
  context-free syntax
    mkdeps-conds(ATerm, ATerm) → ATerm
    makedeps(ATerm, ATerm) → ATerm
    makedeps-pars(ATerm, ATerm, INT) → ATerm
    new-dep(ATerm, ATerm, ATerm, ATerm) → ATerm
equations

The function makedep-condeq derives the attribute dependencies in each equation individually. A string representation is made of the top operator of the left-hand side of the equation. This "string" is annotated with 0 to indicate that it represents a *parent* attribute. The depedencies in the conditions are found by invocation of the function mkdeps-conds. Dependencies in the right-hand side of the equations are derived when this side is applied to the function makedeps.

$$\textsf{makedep-condeq}(T_1;\ T_2) = \textsf{merge}(\textsf{makedep-condeq}(T_1),\ \textsf{makedep-condeq}(T_2)) \quad \text{[DepEq1]}$$

$$\frac{T_4 = \textsf{mkdeps-conds}(T_3,\ \textsf{mk-decl}(T_1)\ /\ \texttt{"0"}),\quad T_5 = \textsf{makedeps}(T_2,\ \textsf{mk-decl}(T_1)\ /\ \texttt{"0"})}{\textsf{makedep-condeq}([\textsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ [L\ T_3]]) = \textsf{merge}(T_4,\ T_5)} \quad \text{[DepEq2]}$$

$$\textsf{makedep-condeq}(T) \ = \ \textsf{nil} \quad \textbf{otherwise} \quad \text{[DepEq]}$$

Below the definition is given for deriving dependencies in conditions. The left- and right-hand side of a condition are applied to the function makedeps.

$$\textsf{mkdeps-conds}(T_1;\ T_2,\ T_3) = \textsf{merge}(\textsf{mkdeps-conds}(T_1,\ T_3),\ \textsf{mkdeps-conds}(T_2,\ T_3)) \quad \text{[DepCond1]}$$

$$\textsf{mkdeps-conds}([\textsf{scCondition}\ L;\ T_1;\ T_2],\ T_3) = \textsf{merge}(\textsf{makedeps}(T_1,\ T_3),\ \textsf{makedeps}(T_2,\ T_3)) \quad \text{[DepCond2]}$$

$$\textsf{mkdeps-conds}(T_1,\ T_2) \ = \ \textsf{nil} \quad \textbf{otherwise} \quad \text{[DepCond]}$$

In equations [MDep1]-[MDep4] below the definition of the function makedeps is given. When the terms for which dependencies must be derived is a list, [MDep1] applies.

$$\mathsf{makedeps}(T_1;\ T_2,\ T_3) = \mathsf{merge}(\mathsf{makedeps}(T_1,\ T_3),\ \mathsf{makedeps}(T_2,\ T_3)) \qquad \text{[MDep1]}$$

Equation [MDep2] applies when the first argument is an incremental term with no parameters. The constructor of this term is annotated with a child number T. A new dependency can be created using the second argument. A dependency runs from the attribute related to the first argument to the attribute in the second argument of the function. This dependency is created by means of the function new-dep.

$$\frac{\mathsf{true} = \mathsf{isInc}(T_3)}{\begin{array}{l}\mathsf{makedeps}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4\ /\ T],\ T_6\ /\ T') = \\ \mathsf{new\text{-}dep}(\mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])),\ T,\ \mathsf{fun\text{-}att}(T_6),\ T')\end{array}} \qquad \text{[MDep2]}$$

Equation [MDep3] applies when the first argument is an incremental term with parameters. A dependency between the function attributes is created by means of the function new-dep. Dependencies for the parameter attributes are made by applying the function makedeps-pars.

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{isInc}(T_3), \\ T_8 = \mathsf{new\text{-}dep}(\mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])),\ T,\ \mathsf{fun\text{-}att}(T_6),\ T'), \\ T_9 = \mathsf{makedeps\text{-}pars}(T_5,\ \mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])\ /\ T,\ 1)\end{array}}{\begin{array}{l}\mathsf{makedeps}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4\ /\ T;\ T_5],\ T_6\ /\ T') = \\ \mathsf{merge}(T_8,\ T_9)\end{array}} \qquad \text{[MDep3]}$$

When the term in the first argument is not incremental, dependencies are make for its parameters by recursively applying the function makedeps.

$$\frac{\mathsf{false} = \mathsf{isInc}(T_3)}{\begin{array}{l}\mathsf{makedeps}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4],\ T_6) = \\ \mathsf{makedeps}(T_4,\ T_6)\end{array}} \qquad \text{[MDep4]}$$

$$\mathsf{makedeps}(T,\ T_1)\ =\ \mathsf{nil} \quad \textbf{otherwise} \qquad \text{[MDep]}$$

Below the definition is given for deriving dependencies between parameters attributes.

$$\mathsf{makedeps\text{-}pars}(T_1;\ T_2,\ T_3,\ Int) = \mathsf{merge}(\mathsf{makedeps\text{-}pars}(T_1,\ T_3,\ Int),\ \mathsf{makedeps\text{-}pars}(T_2,\ T_3,\ Int+1)) \qquad \text{[MDepP1]}$$

$$\mathsf{makedeps\text{-}pars}(T_1\ /\ T,\ T_2\ /\ T',\ Int) = \mathsf{new\text{-}dep}(T,\ \texttt{"0"},\ \mathsf{par\text{-}att}(T_2,\ Int),\ T') \qquad \text{[MDepP2]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{isInc}(T_3), \\ T_7 = \mathsf{new\text{-}dep}(\mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])),\ T,\ \mathsf{par\text{-}att}(T_6,\ Int),\ T'), \\ T_8 = \mathsf{makedeps\text{-}pars}(T_5,\ \mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])\ /\ T,\ 1)\end{array}}{\begin{array}{l}\mathsf{makedeps\text{-}pars}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4\ /\ T;\ T_5],\ T_6\ /\ T',\ Int) = \\ \mathsf{merge}(T_7,\ T_8)\end{array}} \qquad \text{[MDepP3]}$$

$$\frac{\mathsf{true} = \mathsf{isInc}(T_3)}{\begin{array}{l}\mathsf{makedeps\text{-}pars}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4\ /\ T],\ T_6\ /\ T',\ Int) = \\ \mathsf{new\text{-}dep}(\mathsf{fun\text{-}att}(\mathsf{mk\text{-}decl}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3])),\ T,\ \mathsf{par\text{-}att}(T_6,\ Int),\ T')\end{array}} \qquad \text{[MDepP4]}$$

makedeps-pars(T, T_1, Int) = nil **otherwise** [MDepP]

By applying the function new-dep defined below, an AsFix term representing a dependency is created.

```
new-dep(T1, T2, T3, T4) =                                  [NewDep]
 [[scCfFunction
   [scCfElems
    [scSort scATT-NAME];
    [scQLit ":"];
    [scSort scINT];
    [scQLit "->"];
    [scSort scATT-NAME];
    [scQLit ":"];
    [scSort scINT]];
 [scSort "DEP"];
 [scAttributes nil]]
 [scLex T1; [scSort scATT-NAME]];
 [scLex T2; [scSort scINT]];
 [scLex T3; [scSort scATT-NAME]];
 [scLex T4; [scSort scINT]]]
```

G.4.10 AuxInitUpdate

In the module AuxInitUpdate some global auxiliary function have been defined.

module AuxInitUpdate
imports AuxInitPath[G.1.4]
exports
  context-free syntax

```
    init-operators(ATerm)            → ATerm
    init-operators-with-rank(ATerm)  → ATerm
    iu-cond(ATerm, ATerm)            → ATerm
    mk-iter(ATerm)                   → ATerm

    scinit                           → Literal
    scupdate-needed                  → Literal
    scnew-fun-att                    → Literal
    scnew-par-att                    → Literal
    scadd-deps                       → Literal
    scDEP                            → Literal
    scDEPS                           → Literal
    scBool                           → Literal
    scEqDef                          → Literal
```

equations

```
init-operators(T1; T2) =                                   [IO1]
 merge(init-operators(T1), init-operators(T2))

init-operators(T) =                                        [IO]
 [scCfFunction                     otherwise
  [scCfElems scinit; "("; T; ")"];
  [scSort scBOOL];
  [scAttributes nil]]
```

init-operators-with-rank(T_1; T_2) = [IOWR1]
merge(init-operators-with-rank(T_1), init-operators-with-rank(T_2))

init-operators-with-rank(T) = [IOWR]
[scCfFunction **otherwise**
[scCfElems scinit; "("; T; ","; [scSort scINT]; ")"];
[scSort scBOOL];
[scAttributes nil]]

mk-iter(T_1; T_2)	=	merge(mk-iter(T_1), mk-iter(T_2))	[MI1]
mk-iter(nil)	=	nil	[MI2]
mk-iter(L)	=	[scLex L; [scSort "INT"]]	[MI3]

T_2 = iter-decl(scINT, ";", "*")
---
iu-cond(T, T_1) = [UNCond]
[scCondition
scEq;
bool-constant(T);
[[scCfFunction
[scCfElems
scupdate-needed; "("; T_2; ","; [scSort scINT]; ")"];
[scSort scBOOL];
[scAttributes nil]]
[T_2 T_1]; varR ( )]]

The following equations turns the introduced constants into strings.

aterms(scinit)	=	"init"	[AT1]
aterms(scupdate-needed)	=	"update-needed"	[AT2]
aterms(scnew-fun-att)	=	"new-fun-att"	[AT3]
aterms(scnew-par-att)	=	"new-par-att"	[AT4]
aterms(scadd-deps)	=	"add-deps"	[AT5]
aterms(scDEP)	=	"DEP"	[AT6]
aterms(scDEPS)	=	"DEPS"	[AT7]
aterms(scBool)	=	"Bool"	[AT8]

G.5 Derive InitUpdate module, ASF+SDF version

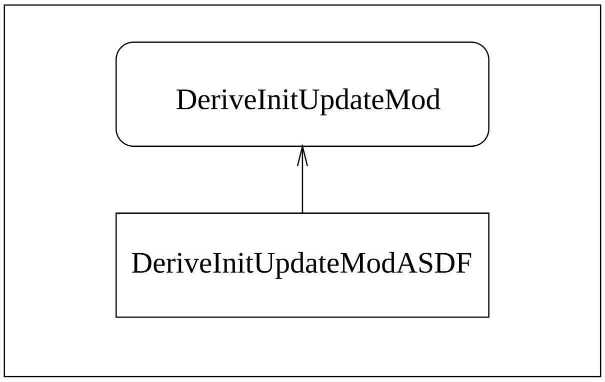


Figure G.5: Module import overview.

G.5.1 DeriveInitUpdateASDF

In the module DeriveInitUpdateASDF the specification of the deriving an InitUpdate module for the ASF+SDF version is given. First the InitUpdate module for the external version is derived an that module is transformed into an ASF+SDF version by adding the storage and changeing the result sorts of the functions.

module DeriveInitUpdateASDF
imports DeriveInitUpdateMod$^{G.4.1}$
exports
  context-free syntax
    derive-init-update-ASDF({ATerm ","}+) → ATerm
hiddens
  context-free syntax
    extc2asdf(ATerm) → ATerm
    extc2asdf(ATerm, ATerm) → ATerm
    extc2asdf-cond(ATerm, ATerm) → ATerm
    add-s-rhs-un(ATerm, ATerm) → ATerm
    add-s-lhs-un(ATerm, ATerm) → ATerm
    varS(ATerm) → ATerm
    addSTORAGE(ATerm) → ATerm
  variables
    Index [*0-9′*]∗ → ATerm
equations

The function derive-init-update-ASDF defined below derives an InitUpdate module for the ASF+SDF version of incremental specifications from a number of modules in AsFix notation. An InitUpdate module for the external version is first derived by means of the function derive-init-update-mod. This function is defined in the module DeriveInitUpdateMod. The InitUpdate module yielded can be transformed into the ASF+SDF version by adding the storage. This transformation is performed when the function extc2asdf is applied.

$$\frac{T = \text{derive-init-update-mod}(Ts^{+})}{\text{derive-init-update-ASDF}(Ts^{+}) = \text{extc2asdf}(T)} \quad \text{[DIUA]}$$

In equation [E2A1] the module is transformed. The parts of the module in AsFix notation that need adjusments is the *PrefixSyntax*, T_2, and the *PrefixEquations*, T_4.

$$\frac{\begin{array}{l} T_2{}' = \mathsf{extc2asdf}(T_2), \\ T_4{}' = \mathsf{extc2asdf}(T_4) \end{array}}{\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scModule}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) = \\ [\mathsf{scModule}\ L_1;\ L_2;\ T_1;\ T_2{}';\ T_3;\ T_4{}'] \end{array}} \quad \text{[E2A1]}$$

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scPrefixSyntax}\ T]) = \\ [\mathsf{scPrefixSyntax}\ \mathsf{extc2asdf}(T)] \end{array} \quad \text{[E2A2]}$$

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scPrefixEquations}\ T]) = \\ [\mathsf{scPrefixEquations}\ \mathsf{extc2asdf}(T)] \end{array} \quad \text{[E2A3]}$$

Equation [E2A4] applies when the part of the module that should be transformed is a list of syntax sections, function declarations or equations.

$$\begin{array}{l} \mathsf{extc2asdf}(T_1;\ T_2) = \\ \mathsf{merge}(\mathsf{extc2asdf}(T_1), \mathsf{extc2asdf}(T_2)) \end{array} \quad \text{[E2A4]}$$

Equation [E2A5] defines the transformation of the list of imported modules in the InitUpdate module. When the InitUpdate module was derived for the external version, the name of the module StorageInterfaceInit was added at the head of this list, T_1 in the equation below. The ASF+SDF version should import another module: SM-init-update. This module is added to the list.

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scImports}\ T_1;\ T_2]) = \\ [\mathsf{scImports}\ [\mathsf{scId}\ \texttt{"SM-init-update"}];\ T_2] \end{array} \quad \text{[E2A5]}$$

The equations [E2A6] and [E2A7] given below defines the transformation of the Hidden and Export sections of a module. These section are transformed by adjustment of their subsections.

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scExports}\ T]) = \\ [\mathsf{scExports}\ \mathsf{extc2asdf}(T)] \end{array} \quad \text{[E2A6]}$$

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scHiddens}\ T]) = \\ [\mathsf{scHiddens}\ \mathsf{extc2asdf}(T)] \end{array} \quad \text{[E2A7]}$$

The context free syntax section is adjusted by transforming the function declarations of the functions that have been declared in it, equation [E2A8].

$$\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scContextFreeSyntax}\ T]) = \\ [\mathsf{scContextFreeSyntax}\ \mathsf{extc2asdf}(T)] \end{array} \quad \text{[E2A8]}$$

Equation [E2A9] defines the transformation of a function declaration. The function storage yields the ATerm representing the sort STORAGE. This sort is added to the list of arguments in the function declaration by means of the function add2cfelems. The function is given the result sort STORAGE. (T_2 is replaced by T, the sort STORAGE).

$$\frac{\begin{array}{l} T_1{}' = \mathsf{addSTORAGE}(T_1), \\ T_2{}' = [\mathsf{scSort}\ \mathsf{scSTORAGE}] \end{array}}{\begin{array}{l} \mathsf{extc2asdf}([\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]) = \\ [\mathsf{scCfFunction}\ T_1{}';\ T_2{}';\ T_3] \end{array}} \quad \text{[E2A9]}$$

In equation [E2A10] below, the definition of the transformation of an equation in ASFIX notation is given. If an equation in the InitUpdate module has conditions, the first condition contains the application of the function update-needed. In the ASF+SDF version, this function has the sort BOOL-STORAGE as its result sort. This first condition must thus be treated different than the other conditions. This is accomplished when the function extc2asdf-cond is applied to the conditions, T_3 in the equations. The variable representing the storage in the left-hand side of an equation is given index 1, the first condition in equations [E2A10]. During the transformation of the conditions, this index might increase thus a new index, *Index* is returned when the conditions have been transformed. With this index a new storage variable S_{Index} becomes the new right-hand side of the equations, condition three or is passed as an argument when the right-hand side of the equations is transformed. Equation [E2A10a] applies when the equation is an update needed equation, these equations always have more than one condition. [E2A10b] applies when it is an update not needed equation having exactly one condition and is not an equation defining an injection. [E2A10c] applies when the equation defines an injection. Only the left- and right-hand side of the equation and [E2A10d] applies when the equation has no condition.

$$\frac{\begin{array}{r@{\;}l} & T_1{}' = \mathsf{extc2asdf}(T_1,\ \texttt{"1"}),\\ Index\ /\ & T_3{}' = \mathsf{extc2asdf\text{-}cond}(T_3;\ T_4,\ \texttt{"1"}),\\ & T_2{}' = \mathsf{extc2asdf}(T_2,\ Index)\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ [L\ T_3;\ T_4]]) =\\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1{}';\ T_2{}';\ [L\ T_3{}']]\end{array}} \quad \text{[E2A10a]}$$

$$\frac{\begin{array}{r@{\;}l} & T_3 = [L'\ L'';\ [\mathsf{scVar}\ T];\ [[\mathsf{scCfFunction}\ T']\ T'']],\\ & T_1{}' = \mathsf{extc2asdf}(T_1,\ \texttt{"1"}),\\ Index\ /\ & T_3{}' = \mathsf{extc2asdf\text{-}cond}(T_3,\ \texttt{"1"}),\\ & T_2{}' = \mathsf{varS}(Index)\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ [L\ T_3]]) =\\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1{}';\ T_2{}';\ [L\ T_3{}']]\end{array}} \quad \text{[E2A10b]}$$

$$\frac{\begin{array}{l} T_3 = [L'\ L'';\ [\mathsf{scVar}\ T'];\ [\mathsf{scVar}\ T'']],\\ T_1{}' = \mathsf{extc2asdf}(T_1,\ \texttt{"1"}),\\ T_2{}' = \mathsf{extc2asdf}(T_2,\ \texttt{"1"})\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ [L\ T_3]]) =\\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1{}';\ T_2{}';\ [L\ T_3]]\end{array}} \quad \text{[E2A10c]}$$

$$\frac{\begin{array}{l} T_1{}' = \mathsf{extc2asdf}(T_1,\ \texttt{"1"}),\\ T_2{}' = \mathsf{extc2asdf}(T_2,\ \texttt{"1"})\end{array}}{\begin{array}{l}\mathsf{extc2asdf}([\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1;\ T_2;\ [L\ \mathsf{nil}]]) =\\ [\mathsf{scCondEquation}\ L_1;\ L_2;\ T_1{}';\ T_2{}';\ [L\ \mathsf{nil}]]\end{array}} \quad \text{[E2A10d]}$$

The default. If the ATerm T did not match any of the cases described above, it is returned unchanged.

$$\mathsf{extc2asdf}(T) \ = \ T \quad \textbf{otherwise} \quad \text{[E2A]}$$

The equation [E2AI1] defines the transformation of a list of ATerms (in this case conditions) when the index must be maintained. The head of the list is transformed in the first condition. Given the index after this transformation, $Index_1$, the rest of the list can be transformed.

$$\frac{\begin{array}{l} Index_1\ /\ T_1{}' = \mathsf{extc2asdf}(T_1,\ Index),\\ Index_2\ /\ T_2{}' = \mathsf{extc2asdf}(T_2,\ Index_1)\end{array}}{\begin{array}{l}\mathsf{extc2asdf}(T_1;\ T_2,\ Index) =\\ Index_2\ /\ \mathsf{merge}(T_1{}',\ T_2{}')\end{array}} \quad \text{[E2AI1]}$$

Equation [E2AI1] given below defines the transformation of a condition. The index of the variable representing the storage on the left-hand side of the condition, T_1, must be increased by one with respect to the index on the right-hand side of the condition, T_2. The function next-var-index increases the index with one.

$$\frac{\begin{array}{r}T_2{}' = \text{extc2asdf}(T_2,\ \mathit{Index}),\\ \mathit{Index}_1 = \text{next-var-index}(\mathit{Index}),\\ T_1{}' = \text{extc2asdf}(T_1,\ \mathit{Index}_1)\end{array}}{\begin{array}{l}\text{extc2asdf}([\text{scCondition } L;\ T_1;\ T_2],\ \mathit{Index}) =\\ \mathit{Index}_1\ /\ [\text{scCondition } L;\ T_1{}';\ T_2{}']\end{array}} \qquad \text{[E2AI2]}$$

Typical for all conditions in the InitUpdate module, the first condition excepted, is that the left-hand side is a variable of sort BOOL. This is also true for the right hand-side of some equations. In the ASF+SDF version of the InitUpdate module, this variable can replaced by a variable of sort STORAGE with proper index. The function storage returns the variable: S_{Index}.

$$\text{extc2asdf}([\text{scVar } T],\ \mathit{Index})\ =\ \text{varS}(\mathit{Index}) \qquad \text{[E2AI3]}$$

The equation [E2AI4] applies when a variable representing the the storage must be added to a term that is a function application. In the first condition the sort STORAGE is added to the list of arguments and in the second condition the variable representing the storage with index *Index* is added to the list of variables in the term.

$$\frac{\begin{array}{r}T' = \text{extc2asdf}([\text{scCfFunction } T]),\\ T_1{}' = \text{merge}(T_1,\ \text{varS}(\mathit{Index}))\end{array}}{\begin{array}{l}\text{extc2asdf}([[\text{scCfFunction } T]\ T_1],\ \mathit{Index}) =\\ [T'\ T_1{}']\end{array}} \qquad \text{[E2AI4]}$$

When no match is found in equation [E2AI1]-[E2AI4], the default equation is applied.

$$\text{extc2asdf}(T,\ \mathit{Index})\ =\ \mathit{Index}\ /\ T \quad \textbf{otherwise} \qquad \text{[E2AI]}$$

The equation [E2IC1] given below applies when the input that should be transformed by means of the function extc2asdf-cond is a list of ATerms representing conditions. The first condition in this list contains the invocation of the function update-needed that should be transformed by application of extc2asdf-cond. The rest of the list is transformed by means of the extc2asdf. The variable *Index* is used to number the variable representing the storage.

$$\frac{\begin{array}{l}\mathit{Index}_1\ /\ T_1{}' = \text{extc2asdf-cond}(T_1,\ \mathit{Index}),\\ \mathit{Index}_2\ /\ T_2{}' = \text{extc2asdf}(T_2,\ \mathit{Index}_1)\end{array}}{\begin{array}{l}\text{extc2asdf-cond}(T_1;\ T_2,\ \mathit{Index}) =\\ \mathit{Index}_2\ /\ \text{merge}(T_1{}',\ T_2{}')\end{array}} \qquad \text{[E2AC1]}$$

Equations [E2IC2] given below applies when the first condition in an equation in the InitUpdate module is transformed. In this first condition the function update-needed is invoked. The right-hand side of this condition is transformed by means of the function add-s-rhs-un, the index is increased and the left-hand side of the condition is transformed when the function add-s-lhs-un is applied to the left-hand side of the original condition, T_1 and the new index Index_1.

$$\frac{\begin{array}{r}T_2{}' = \text{add-s-rhs-un}(T_2,\ \mathit{Index}),\\ \mathit{Index}_1 = \text{next-var-index}(\mathit{Index}),\\ T_1{}' = \text{add-s-lhs-un}(T_1,\ \mathit{Index}_1)\end{array}}{\begin{array}{l}\text{extc2asdf-cond}([\text{scCondition } L;\ T_1;\ T_2],\ \mathit{Index}) =\\ \mathit{Index}_1\ /\ [\text{scCondition } L;\ T_1{}';\ T_2{}']\end{array}} \qquad \text{[E2AC2]}$$

The default equation given below applies when a first condition does not exist.

$$\text{extc2asdf-cond}(T,\ \mathit{Index})\ =\ \mathit{Index}\ /\ T \quad \textbf{otherwise} \qquad \text{[E2AC]}$$

The function add-s-lhs-un defined by equation [ASlhs] given below transforms the right-hand side of the first condition in an equation. The result of the transformation will be a combination of the old left-hand side and the variable representing the storage with the right index: *Bool : S_{index}*.

add-s-lhs-un(T, *Index*) = [ASlhs]
[[scCfFunction
[scCfElems [scSort scBOOL]; [scQLit ":"]; [scSort scSTORAGE]];
[scSort scBOOL-STORAGE];
[scAttributes nil]]
T; varS(*Index*)]

When the equation [ASrhs] is applied, the term update-needed(*Ranks,Rank*) is transformed into the term update-needed(*Ranks,Rank,S_{index}*) wherein the variable S_{index} represents the storage.

$$\frac{T_1{}' = \mathsf{addSTORAGE}(T_1),\quad T_2{}' = [\mathsf{scSort\ scBOOL\text{-}STORAGE}],\quad T_4{}' = \mathsf{merge}(T_4,\ \mathsf{varS}(\mathit{Index}))}{\mathsf{add\text{-}s\text{-}rhs\text{-}un}([[\mathsf{scCfFunction}\ T_1;\ T_2;\ T_3]\ T_4],\ \mathit{Index}) = [[\mathsf{scCfFunction}\ T_1{}';\ T_2{}';\ T_3]\ T_4{}']} \quad \text{[ASrhs]}$$

The function saddSTORAGE defined below returns the AsFix term representing the input, a function declaration, when the STORAGE has been added as an argument.

addSTORAGE(T) = add2cfelems(T, [scSort scSTORAGE]) [S1]
varS(*Index*) = var-term("S", *Index*, scSTORAGE) [S2]

G.6 Generate PathSearch module

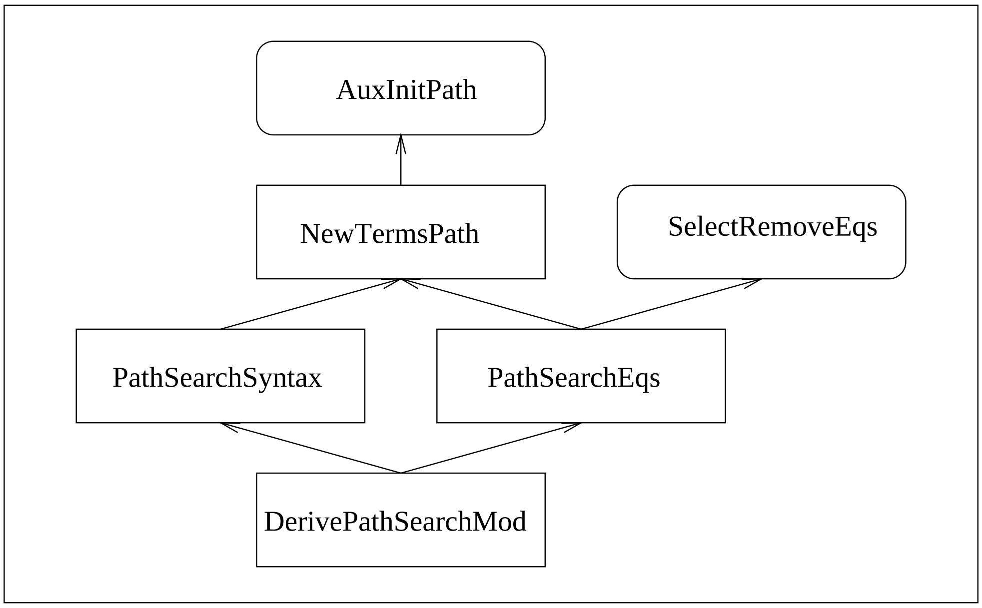


Figure G.6: Module import overview.

G.6.1 DerivePathSearchMod

In the module DerivePathSearchMod the specification of deriving a module PathSearch out of a number of modules in AsFix represention is given.

module DerivePathSearchMod
imports PathSearchSyntax$^{G.6.2}$ PathSearchEqs$^{G.6.3}$
exports
  context-free syntax
    derive-path-search-mod({ATerm ","}*) → ATerm
hiddens
  context-free syntax
    rm-annotations(ATerm) → ATerm
    select-var-sorts(ATerm) → ATerm
    sorts-in(ATerm) → ATerm
equations

Below the definition of the main function for the generation of a module PathSearch is given. In the first condition all equations in the input modules are selected, having incremental top operators on their left-hand sides. The result yielded by the function inc-equations is annotated with the names of the modules in which the equations were found. These annotations are removed when the function rm-annotations is applied. The sorts for which new variables must be defined are selected by means of the function select-var-sorts. The sorts of the constructors are collected when the function constructor-sorts is applied to all declarations of functions that have been declared as incremental. These declarations are selected when inc-operators-in-syntax is applied to the input ATerms representing modules. The injections that can be detected in the selected equations are collected, by the function injections. The name `PathSearch` is given to the generated module, its full name (directory and module name) is given the name: `""`, we do not know in which directory the file will be saved. The SourceSyntax and the SourceEquations are given the value nil. The syntax section and the equations are generated by means of the functions path-search-syntax and path-search-eqs.

$$\frac{\begin{array}{l} T_1 = \text{rm-annotations}(\text{inc-equations}(Ts^+)),\\ T_2 = \text{rmdoubles}(\text{select-var-sorts}(T_1)),\\ T_3 = \text{constructor-sorts}(\text{inc-operators-in-syntax}(Ts^+)),\\ T_4 = \text{rmdoubles}(\text{injections}(T_1)),\\ L_1{}' = \texttt{"PathSearch"},\\ L_2{}' = \texttt{""},\\ T_1{}' = [\text{scSourceSyntax nil}],\\ T_2{}' = \text{path-search-syntax}(T_2,\ T_3,\ T_4),\\ T_3{}' = [\text{scSourceEquations nil}],\\ T_4{}' = \text{path-search-eqs}(T_1) \end{array}}{\begin{array}{l} \text{derive-path-search-mod}(Ts^+) =\\ [\text{scModule } L_1{}';\ L_2{}';\ T_1{}';\ T_2{}';\ T_3{}';\ T_4{}'] \end{array}} \quad \text{[DPSM]}$$

The function rm-annotations defined below removes the annotations of terms in the input list.

$$\text{rm-annotations}(T_1;\ T_2) = \quad \text{[RA1]}$$
$$\text{merge}(\text{rm-annotations}(T_1),\ \text{rm-annotations}(T_2))$$

$$\text{rm-annotations}(T_1\ /\ T_2) \ = \ T_1 \quad \text{[RA2]}$$
$$\text{rm-annotations}(T) \ = \ T \quad \textbf{otherwise} \quad \text{[RA]}$$

The function select-var-sorts defined below selects the sorts of variables in the first argument on the left-hand side of the input equations. When this argument consists of a single variable, the equation defines an injection and the sort is not selected.

$$\text{select-var-sorts}(T_1;\ T_2) = \quad \text{[SvarS1]}$$
$$\text{merge}(\text{select-var-sorts}(T_1),\ \text{select-var-sorts}(T_2))$$
$$\text{select-var-sorts}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = \quad \text{[SvarS2]}$$
$$\text{select-var-sorts}(\text{constructor}(T_1))$$
$$\text{select-var-sorts}([[\text{scCfFunction } T]\ T_1]) = \text{sorts-in}(T_1) \quad \text{[SvarS3]}$$
$$\textbf{when}$$
$$\text{false} = \text{isInjection}([[\text{scCfFunction } T]\ T_1])$$

$$\text{select-var-sorts}(T) \ = \ \text{nil} \quad \textbf{otherwise} \quad \text{[SvarS]}$$

The function sorts-in defined below select all sorts in its input.

$$\text{sorts-in}(T_1;\ T_2) = \quad \text{[Sorts1]}$$
$$\text{merge}(\text{sorts-in}(T_1),\ \text{sorts-in}(T_2))$$

$$\text{sorts-in}(T) \ = \ [\text{scSort sort-of}(T)] \quad \textbf{otherwise} \quad \text{[Sorts]}$$

G.6.2 PathSearchSyntax

In the module PathSearchSyntax the generation of the syntax section for a PathSearch module is specified.

module PathSearchSyntax

imports NewTermsPath$^{G.6.4}$
exports
  context-free syntax
    path-search-syntax(ATerm, ATerm, ATerm) $\rightarrow$ ATerm
hiddens
  context-free syntax
    rm-inj-sorts(ATerm, ATerm) $\rightarrow$ ATerm
    rm(ATerm, ATerm) $\rightarrow$ ATerm
equations

The function path-search-syntax generates the syntax section for a PathSearch module. Input: T_1 are the variable sorts, T_2 are the constructor sorts and T_3 are the injections. Variables are created by means of the function mk-numbered-variables. The variable sorts and the constructor sorts are merged and all doublets in this list are removed and the injection sorts in this list too. The result yielded by the function rm-inj-sorts is used to make new declarations of the sort TERM.

$$\frac{\begin{array}{l} T_4 = \text{mk-numbered-variables}(T_1), \\ T_5 = \text{rm-inj-sorts}(T_3, \text{rmdoubles}(\text{merge}(T_1, T_2))), \\ T_6 = \text{new-TERM-decls}(T_5) \end{array}}{\begin{array}{l} \text{path-search-syntax}(T_1, T_2, T_3) = \\ [\text{scPrefixSyntax} \\ \ [\text{scImports [scId "InitUpdate"]; [scId "AuxPathSearch"]]}; \\ \ [\text{scHiddens} \\ \ \ [\text{scContextFreeSyntax } T_6]; \\ \ \ [\text{scVariables } T_4]]] \end{array}} \quad \text{[PSS]}$$

The function rm-inj-sorts defined below removes all injection sorts, its first argument, out of its second argument, a list of sorts. The injections $T_1 \rightarrow T_2$ are annotated ATerms T_1 / T_2. If both T_1 and T_2 are present in the list of sorts, T_1 is removed out of this list.

$$\frac{\begin{array}{l} \text{true} = \text{occurs}(T_1, T_3), \\ \text{true} = \text{occurs}(T_2, T_3), \\ T_3{}' = \text{rm}(T_1, T_3) \end{array}}{\text{rm-inj-sorts}(T_1 \ / \ T_2, T_3) = T_3{}'} \quad \text{[RMInj1]}$$

$$\frac{\begin{array}{l} \text{true} = \text{occurs}(T_1, T_3), \\ \text{true} = \text{occurs}(T_2, T_3), \\ T_3{}' = \text{rm}(T_1, T_3) \end{array}}{\begin{array}{l} \text{rm-inj-sorts}(T_1 \ / \ T_2; T, T_3) = \\ \text{rm-inj-sorts}(T, T_3{}') \end{array}} \quad \text{[RMInj2]}$$

$$\frac{\text{false} = \text{occurs}(T_1, T_3)}{\begin{array}{l} \text{rm-inj-sorts}(T_1 \ / \ T_2; T, T_3) = \\ \text{rm-inj-sorts}(T, T_3) \end{array}} \quad \text{[RMInj2]}$$

$$\text{rm-inj-sorts}(T, T_1) \ = \ T_1 \quad \textbf{otherwise} \qquad \text{[RMInj]}$$

Below the function rm is defined. It removes its first argument in its second argument.

$$\text{rm}(T, T; T_1) \ = \ T_1 \qquad \text{[RM1]}$$
$$\text{rm}(T, T) \ = \ \text{nil} \qquad \text{[RM2]}$$
$$\text{rm}(T, T_1; T_2) \ = \ \text{merge}(T_1, \text{rm}(T, T_2)) \quad \textbf{when} \ T \neq T_1 \qquad \text{[RM3]}$$
$$\text{rm}(T, T_1) \ = \ T_1 \quad \textbf{when} \ T \neq T_1 \qquad \text{[RM4]}$$

G.6.3 PathSearchEqs

In the module PathSearchEqs the definition of the generation of the equations that should be defined in the module PatSearch is given.

module PathSearchEqs
imports NewTermsPath$^{G.6.4}$ SelectRemoveEqs$^{G.4.4}$
exports
  context-free syntax
    path-search-eqs(ATerm) → ATerm
hiddens
  context-free syntax
    select-constructors(ATerm) → ATerm
    derive-eqs(ATerm, INT) → ATerm
    rename-vars(ATerm, ATerm) → ATerm
    lhs-equation(ATerm) → ATerm
    rhs-equation(ATerm) → ATerm
    path-conditions(ATerm, ATerm, ATerm) → ATerm
    comp-condition(ATerm) → ATerm
    last-condition(ATerm) → ATerm
    path-iter(ATerm, ATerm) → ATerm
equations

The definition of the function path-search-eqs is given below. First the constructors for which equations should be defined are selected by means of the function select-constructors. These constructors can be found in the input, T, containing all equations defining incremental functions. The new equations are generated by applying the function derive-eqs.

$$\frac{T_1 = \text{select-constructors}(T),\quad T_2 = \text{derive-eqs}(T_1, 1)}{\text{path-search-eqs}(T) = [\text{scPrefixEquations } T_2]} \qquad \text{[PSEqs]}$$

The function select-constructors defined below selects the distinct constructors on the left-hand side of the equations in the input, a list of ATerms representing equations. These constructors are only selected when they do not define an injection.

$$\frac{T_1{}' = \text{select-constructors}(T_1),\quad T_2{}' = \text{rm-eqs-same-constructor}(T_1,\ T_2)}{\text{select-constructors}(T_1;\ T_2) = \text{merge}(T_1{}', \text{select-constructors}(T_2{}'))} \qquad \text{[SelCon1]}$$

$$\frac{T = \text{constructor}(T_1),\quad \text{false} = \text{isInjection}(T)}{\text{select-constructors}([\text{scCondEquation } L_1;\ L_2;\ T_1;\ T_2;\ T_3]) = T} \qquad \text{[SelCon2]}$$

$$\text{select-constructors}(T) \ = \ \text{nil} \quad \textbf{otherwise} \qquad \text{[SelCon]}$$

Below the function derive-eqs is defined. The first argument is a list of constructors and the second argument is an integer. This integer will help us to number the tags in the generated equations. A new equations will be generated for for each constructor, equation [DerEqs1]. The input in [DerEqs2] is a constructor: $[[L\ T_1]\ T_2]$ where T_2 represents the variables. These variables are given new names, T_2'. The left-hand side of the equation is created from the constructor with the "old" variables and the constructor and the variables with new names when the function lhs-equation is applied. The conditions in which the function path will be applied are generated when the function path-conditions is invoked. In these conditions a variable P has been given an index. This index is returned, L_1, and will be used when the comp-paths condition is created, function comp-condition, and when the last condition is generated, function last-condition. This index will also be used when the right-hand side of the equation is generated, function rhs-equation. The created components are put in an ATerm representing an equation.

$$\text{derive-eqs}(T_1;\ T_2,\ \mathit{Int}) = \text{merge}(\text{derive-eqs}(T_1,\ \mathit{Int}),\ \text{derive-eqs}(T_2,\ \mathit{Int}+1)) \quad \text{[DerEqs1]}$$

$$\frac{\begin{array}{l} T_2{}' \ /\ T = \text{rename-vars}(T_2,\ \text{nil}),\\ \quad T_3 = \text{lhs-equation}([[L\ T_1]\ T_2];\ [[L\ T_1]\ T_2{}']),\\ L_1 \ /\ T_4 = \text{path-conditions}(T_2,\ T_2{}',\ \texttt{"1"}),\\ \quad T_5 = \text{comp-condition}(L_1),\\ \quad T_6 = \text{last-condition}(L_1),\\ \quad T_7 = \text{rhs-equation}(L_1)\end{array}}{\begin{array}{l}\text{derive-eqs}([[L\ T_1]\ T_2],\ \mathit{Int}) =\\ \text{[scCondEquation}\\ \ \text{scImplies};\\ \ \text{tag}(\texttt{"PS"},\ \mathit{Int});\\ \ T_3;\\ \ T_7;\\ \ \text{[scConditions merge}(T_4,\ \text{merge}(T_5,\ T_6))]]\end{array}} \quad \text{[DerEqs2]}$$

$$\text{derive-eqs}(T,\ \mathit{Int}) \ =\ \text{nil} \quad \textbf{otherwise} \quad \text{[DerEqs]}$$

The function rename-vars defined below renames the variable by using the name of the sort of the variables concatenated with the string "var" and an index will also be given to the variables to distinguish them.

$$\frac{\begin{array}{c} T_1{}' \ /\ T' = \text{rename-vars}(T_1,\ T),\\ T_2{}' \ /\ T'' = \text{rename-vars}(T_2,\ T')\end{array}}{\text{rename-vars}(T_1;\ T_2,\ T) = \text{merge}(T_1{}',\ T_2{}') \ /\ T''} \quad \text{[Rename1]}$$

$$\frac{T_2{}' \ /\ T' = \text{rename-vars}(T_2,\ T)}{\begin{array}{l}\text{rename-vars}([[\text{scCfFunction}\ T_1]\ T_2],\ T) =\\ [[\text{scCfFunction}\ T_1]\ T_2{}'] \ /\ T'\end{array}} \quad \text{[Rename2]}$$

$$\text{rename-vars}([\text{scVar}\ T_1],\ T) \ =\ \text{new-var}(\text{first-sort}(T_1),\ T) \quad \text{[Rename3]}$$

$$\text{rename-vars}(T_1,\ T) \ =\ T_1 \ /\ T \quad \textbf{otherwise} \quad \text{[Rename]}$$

Below the function lhs-equation is defined and yields an ATerm representing the ASF+SDF term path(T,R) where T is the input in equation [lhsEq].

$$\begin{array}{l}\text{lhs-equation}(T) =\\ \text{[path-fun-decl ( )}\\ \ \text{merge}(T,\ \text{varR ( )})]\end{array} \quad \text{[lhsEq]}$$

The function rhs-equation returns an ATerm representing the variables R;P_T, where T is the input and indicates the index the variable P should have.

$$\begin{array}{l}\text{rhs-equation}(T) =\\ \text{[iter-decl(scINT, }\texttt{";"}\text{, }\texttt{"*"}\text{)}\\ \ \text{varR ( ); varP}(T)]\end{array} \quad \text{[rhsEq]}$$

When the function path-conditions is applied new conditions are generated. Its first and second argument are lists representing variables. A condition will be created in which the function path is applied to one variable in the first list and one in the second list: path(T_1,T_1',L) where L is the child number of the variables; first variable has child number 1, the second 2 and so forth. The right-hand side of the created condition will be a variable P_L. The terms [T_1 T_2] and [T_1' T_2'] in equation [PathCond2] represents variables.

$$\frac{L_1 \;/\; T_3 = \text{path-conditions}(T_1,\ T_1',\ L),\quad L_2 \;/\; T_4 = \text{path-conditions}(T_2,\ T_2',\ L_1)}{\text{path-conditions}(T_1;\ T_2,\ T_1';\ T_2',\ L) = L_2 \;/\; \text{merge}(T_3,\ T_4)} \quad \text{[PathCond1]}$$

path-conditions([T_1 T_2], [T_1' T_2'], L) = [PathCond2]
next-var-index(L)
/ [scCondition
scEq;
varP(L);
[path-fun-decl ( )
[T_1 T_2]; [T_1' T_2']; lex-term(L, scINT)]]

path-conditions(T_1, T_2, L) = L / nil **otherwise** [PathCond]

In equation [CompCond] given below the function comp-condition is given. This function creates a condition: P_L = comp-paths(P_1:...:P_{L-1}).

comp-condition(L) = [CompCond]
[scCondition
scEq;
varP(L);
[comp-fun-decl ( )
[iter-decl(scPATH, ":", "*") path-iter("1", L)]]]

Below the function last-condition is defined. This function yields an ATerm representing the condition: $P_L \neq 0$.

last-condition(L) = [LastCond]
[scCondition
scNeq;
varP(L);
lex-term("0", scINT)]

The function path-iter defined below creates a list of ATerms representing the variables $P_{first-argument}$ to $P_{second-argument-1}$.

path-iter(L, L) = nil [PathIter1]

$$\frac{L_1 \neq L_2,\quad L_1' = \text{next-var-index}(L_1)}{\text{path-iter}(L_1,\ L_2) = \text{merge}(\text{varP}(L_1),\ \text{path-iter}(L_1',\ L_2))} \quad \text{[PathIter2]}$$

G.6.4 NewTermsPath

In the module NewTermsPath the definition of ATerms representing ASF+SDF terms are given. These terms are used when a PathSearch module is generated. Some new Literals have also been defined in this module. These Literals are only related to this generation of a module.

module NewTermsPath
imports AuxInitPath $^{G.1.4}$
exports
context-free syntax

```
new-TERM-decls(ATerm) → ATerm
path-fun-decl()       → ATerm
comp-fun-decl()       → ATerm
varP(ATerm)           → ATerm

scTERM                → Literal
scPATH                → Literal
scpath                → Literal
scAND                 → Literal
sccomp-paths          → Literal
```

equations

The function new-TERM-decls defined below returns a list of ATerms representing the declaration of new TERMs. For an input sort SORT, the declaration SORT → TERM is returned.

new-TERM-decls(T_1; T_2) = [NTERMd1]
merge(new-TERM-decls(T_1), new-TERM-decls(T_2))

new-TERM-decls([T T_1]) = [NTERMd2]
[scCfFunction
[scCfElems [T T_1]];
[scSort scTERM];
[scAttributes nil]]

The function path-fun-decl returns an ATerm representing the function declaration:
path(TERM,TERM,INT) → PATH.

path-fun-decl ( ) = [PFD]

```
[scCfFunction
 [scCfElems
  [scQLit scpath];
  [scSort scTERM];
  [scQLit scAND];
  [scSort scTERM];
  [scQLit ","];
  [scSort scINT];
  [scQLit ")"]];
[scSort scPATH];
[scAttributes nil]]
```

The function comp-fun-decl returns an ATerm representing the function declaration:
comp-paths({PATH ":"}*) → PATH.

comp-fun-decl ( ) = [CFD]

[scCfFunction
[scCfElems
sccomp-paths; "("; iter-decl(scPATH, ":", "*"); ")"];
[scSort scPATH];
[scAttributes nil]]

The function varP returns an ATerm representing the variable P_L.

varP(L) = [VarP]
[scVar cc("P"; L); [scSort scPATH]]

The equations below converts the introduced literals into strings.

aterms(scTERM)	=	"TERM"	[AT1]
aterms(scpath)	=	"path("	[AT2]
aterms(scPATH)	=	"PATH"	[AT3]
aterms(sccomp-paths)	=	"comp-paths"	[AT4]
aterms(scAND)	=	"AND"	[AT5]

Appendix H

Storage management modules

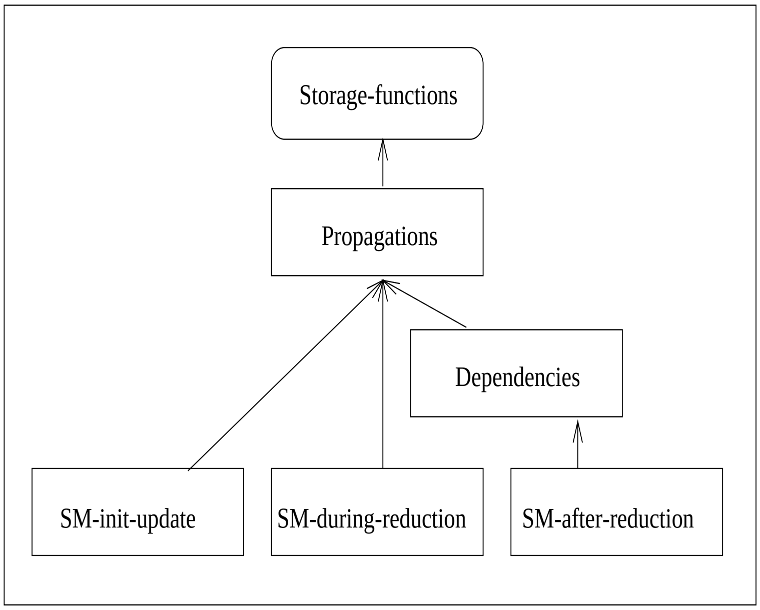


Figure H.1: Module import overview.

H.1 Propagations

In the module Propagations we have specified the propgations that are used to reset the status of the attributes in the storage before, during and after incremental reduction.

module Propagations
imports Storage-functions$^{F.6}$
exports
  context-free syntax
    prop-te-up(ATT-PTRS, ATTRIBUTES) $\rightarrow$ ATTRIBUTES
    prop-te-down(ATT-PTRS, ATTRIBUTES) $\rightarrow$ ATTRIBUTES
    prop-unchanged(ATT-PTRS, ATTRIBUTES) $\rightarrow$ ATTRIBUTES
    prop-non-inc-child(NODE, ATTRIBUTES) $\rightarrow$ ATTRIBUTES
    reset-changed(ATT-PTRS, ATTRIBUTES) $\rightarrow$ ATTRIBUTES

    set-status(STATUS, ATT-PTRS, ATTRIBUTES) $\rightarrow$ ATTRIBUTES set-status(STATUS, {INT ";"}*, ATTRIBUTES) → ATTRIBUTES

hiddens

context-free syntax

prop-te-up-succs(ATT-PTRS, ATTRIBUTES) → ATTRIBUTES
prop-te-up-succ(ATTRIBUTE, ATTRIBUTES) → ATTRIBUTES
prop-te-down-succs(ATT-PTRS, ATTRIBUTES) → ATTRIBUTES
prop-unchanged-succs(ATT-PTRS, ATTRIBUTES) → ATTRIBUTES
reset-changed-succs(ATT-PTRS, ATTRIBUTES) → ATTRIBUTES
all-have-status(STATUS, ATT-PTRS, ATTRIBUTES) → BOOL

equations

The function prop-te-up defined below selects the successors of the function attribute refered to by the pointer *AP*. These successors are further propagated by means of the function prop-te-up-succs.

$$\text{prop-te-up}(, As) \;=\; As \qquad \text{[TEUp1]}$$

$$\frac{\begin{array}{l} APs_1 = \text{succs-of}(\text{AP} : \text{AC}), \\ As = \text{prop-te-up-succs}(APs_1, As_1; As_2; \text{AP} : \text{AC}) \end{array}}{\begin{array}{l} \text{prop-te-up}(\text{AP}; APs, As_1; \text{AP} : \text{AC}; As_2) = \\ \text{prop-te-up}(APs, As) \end{array}} \qquad \text{[TEUp2]}$$

The function prop-te-up-succs defined by equation [TEUss1] - [TEUss3] given below, first determines whether the attribute is safe or not. With safe we mean that the status of the attribute is either C (Changed) or UC (UnChanged), otherwise the attribute is not safe. If the attribute is safe ([TEUss2]), its status will be set to TE (TobeEvaluated) and the function prop-te-up-succ will be applied with the attribute as argument. If the status of the attribute is not safe ([TEUss3]), no action is taken with the attribute.

$$\text{prop-te-up-succs}(, As) \;=\; As \qquad \text{[TEUss1]}$$

$$\frac{\begin{array}{r} \text{true} = \text{safe-att}(\text{AP} : \text{AC}), \\ A = \text{set-status}(\text{TE}, \text{AP} : \text{AC}), \\ As = \text{prop-te-up-succ}(A, As_1; As_2; A) \end{array}}{\begin{array}{l} \text{prop-te-up-succs}(\text{AP}; APs, As_1; \text{AP} : \text{AC}; As_2) = \\ \text{prop-te-up-succs}(APs, As) \end{array}} \qquad \text{[TEUss2]}$$

$$\frac{\text{false} = \text{safe-att}(\text{AP} : \text{AC})}{\begin{array}{l} \text{prop-te-up-succs}(\text{AP}; APs, As_1; \text{AP} : \text{AC}; As_2) = \\ \text{prop-te-up-succs}(APs, As_1; \text{AP} : \text{AC}; As_2) \end{array}} \qquad \text{[TEUss3]}$$

The function prop-te-up-succ defined by equation [TEUs1] and [TEUs2] given below determines whether the attribute is a function attribute, or a parameter attribute. If the attribute is a function attribute, a prop-te-up is performed with this attribute. If the attribute is a parameter attribute, its corresponding function attributes will be selected. The status of these corresponding function attributes is set to TE (TobeEvaluated) and the function prop-te-up is applied to these corresponding function attributes.

$$\frac{\text{fun} = \text{type-of}(\text{AP} : \text{AC})}{\text{prop-te-up-succ}(\text{AP} : \text{AC}, As) = \text{prop-te-up}(\text{AP}, As)} \qquad \text{[TEUs1]}$$

$$\frac{\begin{array}{r} \text{par} = \text{type-of}(A), \\ APs = \text{corr-f-atts-of}(A), \\ As_1 = \text{set-status}(\text{TE}, APs, As) \end{array}}{\text{prop-te-up-succ}(A, As) = \text{prop-te-up}(APs, As_1)} \qquad \text{[TEUs2]}$$

The function prop-te-down defined below first determines whether an attribute is a parameter attribute or a function attribute. If it is a function attribute, Equation [TED2], the status of all predecessors of the attribute is set to TE (TobeEvaluated) and the function prop-te-up is applied for the same function attribute. If the attribute is a parameter attribute, equation [TED3], the function prop-te-down-succs is applied for all the successors of the parameter attribute.

$$\text{prop-te-down}(, \mathit{As}) \;=\; \mathit{As} \qquad [\text{TED1}]$$

$$\frac{\begin{array}{r@{}l} \text{fun} &= \text{type-of}(\text{AP} : \text{AC}), \\ \mathit{APs}_1 &= \text{preds-of}(\text{AP} : \text{AC}), \\ \mathit{As} &= \text{set-status}(\text{TE}, \mathit{APs}_1, \mathit{As}_1; \mathit{As}_2; \text{AP} : \text{AC}), \\ \mathit{As}_3 &= \text{prop-te-up}(\text{AP}, \mathit{As}) \end{array}}{\begin{array}{l} \text{prop-te-down}(\text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{prop-te-down}(\mathit{APs}, \mathit{As}_3) \end{array}} \qquad [\text{TED2}]$$

$$\frac{\text{fun} \neq \text{type-of}(\text{AP} : \text{AC})}{\begin{array}{l} \text{prop-te-down}(\text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{prop-te-down}(\mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) \end{array}} \qquad [\text{TED3}]$$

The function prop-te-down-succs defined below checks whether the attribute is a parameter attribute and also determines if the attribute has status C (Changed) or UC (UnChanged). If these conditions both are ful-filled, the status of the attribute is set to TE (TobeEvaluated) and the function prop-te-up is applied to all the corresponding function attributes that it has.

$$\text{prop-te-down-succs}(, \mathit{As}) \;=\; \mathit{As} \qquad [\text{TEDs1}]$$

$$\frac{\begin{array}{r@{}l} \text{par} &= \text{type-of}(\text{AP} : \text{AC}), \\ \text{true} &= \text{safe-att}(\text{AP} : \text{AC}), \\ \mathit{A} &= \text{set-status}(\text{TE}, \text{AP} : \text{AC}), \\ \mathit{APs}_1 &= \text{corr-f-atts-of}(\mathit{A}), \\ \mathit{As} &= \text{prop-te-up}(\mathit{APs}_1, \mathit{As}_1; \mathit{As}_2; \mathit{A}) \end{array}}{\begin{array}{l} \text{prop-te-down-succs}(\text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{prop-te-down-succs}(\mathit{APs}, \mathit{As}) \end{array}} \qquad [\text{TEDs2}]$$

$$\begin{array}{l} \text{prop-te-down-succs}(\text{AP}; \mathit{APs}, \mathit{As}) = \\ \text{prop-te-down-succs}(\mathit{APs}, \mathit{As}) \quad \textbf{otherwise} \end{array} \qquad [\text{TEDs}]$$

The function prop-unchanged defined by the equations [UC1] and [UC2], selects the successors of an attribute and applies the function prop-unchanged-succs to these successors.

$$\text{prop-unchanged}(, \mathit{As}) \;=\; \mathit{As} \qquad [\text{UC1}]$$

$$\frac{\begin{array}{r@{}l} \mathit{APs}_1 &= \text{succs-of}(\text{AP} : \text{AC}), \\ \mathit{As} &= \text{prop-unchanged-succs}(\mathit{APs}_1, \mathit{As}_1; \mathit{As}_2; \text{AP} : \text{AC}) \end{array}}{\begin{array}{l} \text{prop-unchanged}(\text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{prop-unchanged}(\mathit{APs}, \mathit{As}) \end{array}} \qquad [\text{UC2}]$$

The function prop-unchanged-succs defined below sets the status of an attribute to UC (UnChanged) and applies the function prop-unchanged to this attribute under the condition that the attribute has status TE (TobeEvaluated) and that all its predecessors have status UC. If these conditions are not fulfilled, the attributes stays the same (the default).

$$\text{prop-unchanged-succs}(, \mathit{As}) \;=\; \mathit{As} \qquad [\text{UCs1}]$$

$$\frac{\begin{array}{r}\mathsf{TE} = \mathsf{status\text{-}of}(\mathsf{AP} : \mathsf{AC}),\\ \mathit{APs}_1 = \mathsf{preds\text{-}of}(\mathsf{AP} : \mathsf{AC}),\\ \mathsf{true} = \mathsf{all\text{-}have\text{-}status}(\mathsf{UC}, \mathit{APs}_1, \mathit{As}_1; \mathit{As}_2),\\ \mathit{A} = \mathsf{set\text{-}status}(\mathsf{UC}, \mathsf{AP} : \mathsf{AC}),\\ \mathit{As} = \mathsf{prop\text{-}unchanged}(\mathsf{AP}, \mathit{As}_1; \mathit{As}_2; \mathit{A})\end{array}}{\begin{array}{l}\mathsf{prop\text{-}unchanged\text{-}succs}(\mathsf{AP}; \mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2) =\\ \mathsf{prop\text{-}unchanged\text{-}succs}(\mathit{APs}, \mathit{As})\end{array}} \quad [\mathsf{UCs2}]$$

$$\mathsf{prop\text{-}unchanged\text{-}succs}(\mathsf{AP}; \mathit{APs}, \mathit{As}) = \mathsf{prop\text{-}unchanged\text{-}succs}(\mathit{APs}, \mathit{As}) \quad \textbf{otherwise} \quad [\mathsf{UCs}]$$

The function reset-changed defined below sets the status of attributes to UC (UnChanged) and selects the successors to which the function reset-changed-succs is applied.

$$\mathsf{reset\text{-}changed}(, \mathit{As}) = \mathit{As} \quad [\mathsf{RC1}]$$

$$\frac{\begin{array}{r}\mathit{A} = \mathsf{set\text{-}status}(\mathsf{UC}, \mathsf{AP} : \mathsf{AC}),\\ \mathit{APs}_1 = \mathsf{succs\text{-}of}(\mathit{A}),\\ \mathit{APs}_2 = \mathsf{corr\text{-}f\text{-}atts\text{-}of}(\mathit{A}),\\ \mathit{As} = \mathsf{reset\text{-}changed\text{-}succs}(\mathit{APs}_1; \mathit{APs}_2, \mathit{As}_1; \mathit{As}_2; \mathit{A})\end{array}}{\begin{array}{l}\mathsf{reset\text{-}changed}(\mathsf{AP}; \mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2) =\\ \mathsf{reset\text{-}changed}(\mathit{APs}, \mathit{As})\end{array}} \quad [\mathsf{RC2}]$$

Below we have defined the function reset-changed-succs. Equation [RCs2]: The attribute is a function attribute and its status is set to IN (Initial) and the function prop-te-up is applied to this attribute. Equation [RCs3]: The attribute is a parameter attribute. The status of this attribute and all its corresponding function attributes is set to IN (Initial) and the function prop-te-up is applied to these corresponding function attributes.

$$\mathsf{reset\text{-}changed\text{-}succs}(, \mathit{As}) = \mathit{As} \quad [\mathsf{RCs1}]$$

$$\frac{\begin{array}{r}\mathsf{TE} = \mathsf{status\text{-}of}(\mathsf{AP} : \mathsf{AC}),\\ \mathit{A} = \mathsf{set\text{-}status}(\mathsf{IN}, \mathsf{AP} : \mathsf{AC})\end{array}}{\begin{array}{l}\mathsf{reset\text{-}changed\text{-}succs}(\mathsf{AP}; \mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2) =\\ \mathsf{reset\text{-}changed\text{-}succs}(\mathit{APs}, \mathit{As}_1; \mathit{As}_2; \mathit{A})\end{array}} \quad [\mathsf{RCs2}]$$

$$\frac{\mathsf{TE} \neq \mathsf{status\text{-}of}(\mathsf{AP} : \mathsf{AC})}{\begin{array}{l}\mathsf{reset\text{-}changed\text{-}succs}(\mathsf{AP}; \mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2) =\\ \mathsf{reset\text{-}changed\text{-}succs}(\mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2)\end{array}} \quad [\mathsf{RCs3}]$$

The function prop-non-inc-child sets the status TE in all the function attibutes of the node with node number *NP*. The function prop-te-up is applied to all these function attributes. All attributes that are attached to the children of this node are set to TE by means of the function set-status.

$$\frac{\begin{array}{r}\mathit{APs} = \mathsf{fun\text{-}atts}(\mathsf{NP}, \mathit{As}),\\ \mathit{As}_1 = \mathsf{set\text{-}status}(\mathsf{IN}, \mathit{APs}, \mathit{As}),\\ \mathit{NPs} = \mathsf{children\text{-}of}(\mathsf{NP} : \mathsf{NC}),\\ \mathit{As}_2 = \mathsf{set\text{-}status}(\mathsf{IN}, \mathit{NPs}, \mathit{As}_1),\\ \mathit{As}_3 = \mathsf{prop\text{-}te\text{-}up}(\mathit{APs}, \mathit{As}_2)\end{array}}{\mathsf{prop\text{-}non\text{-}inc\text{-}child}(\mathsf{NP} : \mathsf{NC}, \mathit{As}) = \mathit{As}_3} \quad [\mathsf{PNIC1}]$$

The function set-status defined below sets the status of all attributes pointed at by the list of pointer to *Status*.

$$\frac{\mathit{A} = \mathsf{set\text{-}status}(\mathit{Status}, \mathsf{AP} : \mathsf{AC})}{\begin{array}{l}\mathsf{set\text{-}status}(\mathit{Status}, \mathsf{AP}; \mathit{APs}, \mathit{As}_1; \mathsf{AP} : \mathsf{AC}; \mathit{As}_2) =\\ \mathsf{set\text{-}status}(\mathit{Status}, \mathit{APs}, \mathit{As}_1; \mathit{As}_2; \mathit{A})\end{array}} \quad [\mathsf{SS1}]$$

$$\text{set-status}(\mathit{Status}, \mathit{APs}, \mathit{As}) = \mathit{As} \quad \textbf{otherwise} \qquad \text{[SS]}$$

The function set-status defined below sets the status *Status* in all attributes that are attached to one on the nodes is the list of node pointers (node numbers).

$$\frac{A = \text{set-status}(\mathit{Status}, \text{NP} : \mathit{ANs} : \text{AC})}{\begin{array}{l}\text{set-status}(\mathit{Status}, \mathit{NPs}_1; \text{NP}; \mathit{NPs}_2, \mathit{As}_1; \text{NP} : \mathit{ANs} : \text{AC}; \mathit{As}_2) = \\ \text{set-status}(\mathit{Status}, \mathit{NPs}_1; \text{NP}; \mathit{NPs}_2, \mathit{As}_1; \mathit{As}_2; A)\end{array}} \qquad \text{[SST]}$$

$$\text{set-status}(\mathit{Status}, \mathit{NPs}, \mathit{As}) = \mathit{As} \quad \textbf{otherwise} \qquad \text{[SST]}$$

The function all-have-status defined below is a boolean function yielding true is all attributes refered to by the list of pointers have status *Status*, otherwise it returns false

$$\text{all-have-status}(\mathit{Status}, , \mathit{As}) = \text{true} \qquad \text{[AHS1]}$$

$$\frac{\mathit{Status} = \text{status-of}(\text{AP} : \text{AC})}{\begin{array}{l}\text{all-have-status}(\mathit{Status}, \text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{all-have-status}(\mathit{Status}, \mathit{APs}, \mathit{As}_1; \mathit{As}_2; \text{AP} : \text{AC})\end{array}} \qquad \text{[AHS2]}$$

$$\frac{\mathit{Status} \neq \text{status-of}(\text{AP} : \text{AC})}{\begin{array}{l}\text{all-have-status}(\mathit{Status}, \text{AP}; \mathit{APs}, \mathit{As}_1; \text{AP} : \text{AC}; \mathit{As}_2) = \\ \text{false}\end{array}} \qquad \text{[AHS3]}$$

H.2 SM-during-reduction

In the module SM-reduce (Storage Management during reduction) we have defined the functions that are called to invoke the storage some way.

module SM-during-reduction
imports Propagations$^{H.1}$
exports
  context-free syntax
    status-is-safe(ATT-NAME, INCINFO, STORAGE) → BOOL-STORAGE
    status-is-unsafe(ATT-NAME, INCINFO, STORAGE) → BOOL-STORAGE
    retrieve(ATT-NAME, STORAGE) → NF-STORAGE

    store-par-att(ATT-NAME, NORMAL-FORM, STORAGE) → BOOL-STORAGE
    store-fun-att(ATT-NAME, NORMAL-FORM, STORAGE) → NF-STORAGE
hiddens
  context-free syntax
    set-new-cur(INCINFO, S-DIRECTIONS, NODES) → S-DIRECTIONS
equations

The function set-new-cur defined below sets the new current node in the storage *S*. If the stack in the storage is empty we are at the very begining of a reduction, and we can let the top of the storage be the new current node. If the stack is not empty we will select the top of the stack and chose the rank'th of its children as the new current.

$$\frac{\begin{array}{r}\text{set-level}(0, \text{SD}) = \text{SD}_1, \\ \text{set-cur}(\text{top-of}(\text{SD}_1), \text{SD}_1) = \text{SD}_2, \\ \text{set-stack}(, \text{SD}_2) = \text{SD}_3\end{array}}{\text{set-new-cur}(R : 0, \text{SD}, \mathit{Ns}) = \text{SD}_3} \qquad \text{[SNC1]}$$

$$\frac{\begin{array}{r}L \neq 0,\\ \text{level-of(SD)} = L,\\ N;\ Ns_1 = \text{find(top-of-stack(SD)}, Ns),\\ \text{NP} = \text{select-child-ptr}(R, N),\\ \text{SD}_1 = \text{set-cur(NP, SD)}\end{array}}{\text{set-new-cur}(R : L, \text{SD}, Ns) = \text{SD}_1} \quad \text{[SNC2]}$$

$$\frac{\begin{array}{r}L \neq \text{level-of(SD)},\\ \text{SD}_1 = \text{pop-of-stack(SD)}\end{array}}{\begin{array}{l}\text{set-new-cur}(R : L, \text{SD}, Ns) =\\ \text{set-new-cur}(R : L, \text{SD}_1, Ns)\end{array}} \quad \text{[SNC3]}$$

The function status-of defined below checks if the status of an attribute is safe or not and returns the status safe or unsafe together with a new copy of the storage in which a new current node has been set.

$$\frac{\begin{array}{r}\text{SD}_1 = \text{set-new-cur}(R : L, \text{SD}, Ns),\\ A;\ As_1 = \text{find(cur-of(SD}_1\text{) : AN}, As),\\ \text{true} = \text{safe-att}(A)\end{array}}{\text{status-is-safe(AN}, R : L, \text{SD} : Ns : As) = \text{true} : \text{SD}_1 : Ns : As} \quad \text{[SiS]}$$

$$\frac{\begin{array}{r}\text{SD}_1 = \text{set-new-cur}(R : L, \text{SD}, Ns),\\ A;\ As_1 = \text{find(cur-of(SD}_1\text{) : AN}, As),\\ \text{false} = \text{safe-att}(A),\\ \text{SD}_2 = \text{push-cur-on-stack(SD}_1)\end{array}}{\text{status-is-unsafe(AN}, R : L, \text{SD} : Ns : As) = \text{true} : \text{SD}_2 : Ns : As} \quad \text{[SiU]}$$

The function return-value returns the normal form stored in the attribute that has the name *Att-name*. In which node this attribute can be found is dependent on the current node that is set in the storage *S*.

$$\frac{\begin{array}{r}A;\ As_1 = \text{find(cur-of(SD) : AN}, As),\\ Nf = \text{nf-of}(A)\end{array}}{\text{retrieve(AN, SD} : Ns : As) = Nf : \text{SD} : Ns : As} \quad \text{[Ret1]}$$

The function check-store-par-att definend below checks if the status of the attribute with *Att-name* is safe or unsafe. If it is safe, true is returned together with the storage *S*. If the status of the attribute is unsafe, the normal form *Nf* is stored by means of the function store-par-att and also in this case true is returned together with the storage *S1* that is the result of the function store-par-att; an updated version of *S*.

$$\frac{\begin{array}{r}\text{NP} = \text{cur-of(SD)},\\ A;\ As_1 = \text{find(NP : AN}, As),\\ \text{C} \neq \text{status-of}(A),\\ Nf = \text{nf-of}(A),\\ \text{AP : AC} = \text{set-status(UC}, A),\\ As_2 = \text{prop-unchanged(AP}, As_1;\ \text{AP : AC)}\end{array}}{\text{store-par-att(AN}, Nf, \text{SD} : Ns : As) = \text{true} : \text{SD} : Ns : As_1;\ \text{AP : AC}} \quad \text{[SPA1]}$$

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} &= \mathsf{cur\text{-}of}(\mathsf{SD}),\\ A;\, As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\, As),\\ \mathsf{C} &\neq \mathsf{status\text{-}of}(A),\\ Nf &\neq \mathsf{nf\text{-}of}(A),\\ A_1 &= \mathsf{set\text{-}nf}(Nf,\, A),\\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{C},\, A_1),\\ \mathsf{SD}_1 &= \mathsf{add\text{-}changed\text{-}att}(\mathsf{AP},\, \mathsf{SD}),\\ As_2 &= \mathsf{prop\text{-}te\text{-}down}(\mathsf{AP},\, As_1;\, \mathsf{AP} : \mathsf{AC}) \end{array}}{\mathsf{store\text{-}par\text{-}att}(\mathsf{AN},\, Nf,\, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_1 : Ns : As_1;\, \mathsf{AP} : \mathsf{AC}} \quad [\mathsf{SPA2}]$$

$$\mathsf{store\text{-}par\text{-}att}(\mathsf{AN},\, Nf,\, S) \;=\; \mathsf{true} : S \quad \textbf{otherwise} \qquad [\mathsf{SPA}]$$

Below we have defined the function store-fun-att that stores the normal form in a function attribute. At this point during the reduction we can't relay on the current node pointer in the storage but the node in which this attribute should be stored is at this point on top of the stack in the storage. Dependent on the normal form already stored in the attribute, the normal form *Nf* is stored and a pointer to the the attribute is added to the list of changed attributes in the storage $[sf1]$ or *Nf* is not stored and a propagate-unchanged is performed. In both cases, the last action is to remove the top of the stack in the storage $[sf2]$. This node has now been evaluated.

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ A;\, As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\, As),\\ Nf &\neq \mathsf{nf\text{-}of}(A),\\ A_1 &= \mathsf{set\text{-}nf}(Nf,\, A),\\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{C},\, A_1),\\ \mathsf{SD}_1 &= \mathsf{add\text{-}changed\text{-}att}(\mathsf{AP},\, \mathsf{SD}),\\ \mathsf{SD}_2 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}_1),\\ \mathsf{SD}_3 &= \mathsf{set\text{-}cur}(\mathsf{NP},\, \mathsf{SD}_2) \end{array}}{\mathsf{store\text{-}fun\text{-}att}(\mathsf{AN},\, Nf,\, \mathsf{SD} : Ns : As) = Nf : \mathsf{SD}_3 : Ns : As_1;\, \mathsf{AP} : \mathsf{AC}} \quad [\mathsf{SF1}]$$

$$\frac{\begin{array}{r@{\;}l} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ A;\, As_1 &= \mathsf{find}(\mathsf{NP} : \mathsf{AN},\, As),\\ Nf &= \mathsf{nf\text{-}of}(A),\\ \mathsf{AP} : \mathsf{AC} &= \mathsf{set\text{-}status}(\mathsf{UC},\, A),\\ As_2 &= \mathsf{prop\text{-}unchanged}(\mathsf{AP},\, As_1;\, \mathsf{AP} : \mathsf{AC}),\\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}),\\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP},\, \mathsf{SD}_1) \end{array}}{\mathsf{store\text{-}fun\text{-}att}(\mathsf{AN},\, Nf,\, \mathsf{SD} : Ns : As) = Nf : \mathsf{SD}_2 : Ns : As_2} \quad [\mathsf{SF2}]$$

H.3 SM-after-reduction

In the module SM-after-reduct we have defined one function that should be called after a reduction and before the next one. This function resets all attributes that have status Changed, and gives the status UnChanged.

module SM-after-reduction
imports Propagations$^{H.1}$
exports
  context-free syntax
    reset-changed-atts(STORAGE) $\rightarrow$ STORAGE
equations

The function reset-changed-atts defined below should be called between reductions. This function selects all attributes in the storage that have status Changed, and changes their status to UnChanged by means of the function funreset-changed that is defined in the module Propagations.

$$\frac{\begin{array}{r}\text{ch-atts-of}(\text{SD}) = \mathit{APs},\\ \mathit{As}_1 = \text{reset-changed}(\mathit{APs}, \mathit{As}),\\ \text{SD}_1 = \text{reset-ch-atts}(\text{SD})\end{array}}{\text{reset-changed-atts}(\text{SD} : \mathit{Ns} : \mathit{As}) = \text{SD}_1 : \mathit{Ns} : \mathit{As}_1} \quad \text{[RCA]}$$

H.4 Dependencies

In the module Dependencies, we have defined the dependecies and function for adding the dependecies to a node and its children.

module Dependencies
imports Propagations$^{H.1}$
exports
  sorts DEP DEPS SUN
  context-free syntax
    ATT-NAME ":" INT "→" ATT-NAME ":" INT → DEP
    {DEP "."}* → DEPS
    add-new-deps(NODE, {DEP "."}*, ATTRIBUTES) → ATTRIBUTES
    add-trans-deps(NODE, ATTRIBUTES) → ATTRIBUTES
  variables
    Dep [0-9′]* → DEP
    Deps [0-9′]* → {DEP "."}*
hiddens
  context-free syntax
    add-dep(DEP, NODE-PTRS, ATTRIBUTES) → ATTRIBUTES
    sub-rank-by-nodeptr(INT, NODE-PTRS) → NODE-PTR
    add-dep2atts(ATTRIBUTE, ATTRIBUTE) → ATTRIBUTES
    corr-att(ATT-NAME, ATT-NAME) → BOOL
    corr-att(ATT-NAME, ATT-NAME, ATT-NAME) → BOOL
  variables
    Chars [0-9′]* → CHAR*
equations

The function add-new-deps defined belo adds new dependencies between its attributes and the attributes attached to its children recursively. The children of the node are selected and the function add-dep is invoked with one dependency *Dep* and the node numbers of the children.

$$\text{add-new-deps}(\text{NP} : \text{NC}, , \mathit{As}) \;=\; \mathit{As} \quad \text{[AND1]}$$

$$\frac{\begin{array}{r}\mathit{NPs} = \text{children-of}(\text{NP} : \text{NC}),\\ \mathit{As}_1 = \text{add-dep}(\mathit{Dep}, \text{NP}; \mathit{NPs}, \mathit{As})\end{array}}{\begin{array}{l}\text{add-new-deps}(\text{NP} : \text{NC}, \mathit{Dep}.\ \mathit{Deps}, \mathit{As}) =\\ \text{add-new-deps}(\text{NP} : \text{NC}, \mathit{Deps}, \mathit{As}_1)\end{array}} \quad \text{[AND2]}$$

The function add-dep defined by equation [AD] starts by substituting the ranks in the dependency with the corresponding node number by means of the function sub-rank-by-nodeptr. The attributes that corresponds to the attributes in the dependency are selected and the new successor and predecessor is added to these attributes when the function add-dep2atts is applied.

$$\frac{\begin{array}{r@{\ }l} \mathsf{NP}_1 &= \text{sub-rank-by-nodeptr}(\mathit{Int}_1,\ \mathit{NPs}),\\ \mathsf{NP}_2 &= \text{sub-rank-by-nodeptr}(\mathit{Int}_2,\ \mathit{NPs}),\\ A_1;\ \mathit{As}_1 &= \text{find}(\mathsf{NP}_1 : \mathsf{AN}_1,\ \mathit{As}),\\ A_2;\ \mathit{As}_2 &= \text{find}(\mathsf{NP}_2 : \mathsf{AN}_2,\ \mathit{As}_1),\\ \mathit{As}_3 &= \text{add-dep2atts}(A_1,\ A_2) \end{array}}{\text{add-dep}(\mathsf{AN}_1 : \mathit{Int}_1 \rightarrow \mathsf{AN}_2 : \mathit{Int}_2,\ \mathit{NPs},\ \mathit{As}) = \mathit{As}_3;\ \mathit{As}_2} \quad \text{[AD]}$$

Below the function sub-rank-by-nodeptr is defined. This function returns the node number of the node in which an attribute should get a new successor of predesessor, given the rank in the dependency and a list containg the number of a node followed by the node numbers of its children.

$$\text{sub-rank-by-nodeptr}(0,\ \mathsf{NP};\ \mathit{NPs}) \ = \ \mathsf{NP} \quad \text{[SRBNp1]}$$

$$\text{sub-rank-by-nodeptr}(\mathit{Int},\ \mathsf{NP};\ \mathit{NPs}) \ = \ \text{sub-rank-by-nodeptr}(\mathit{Int} - 1,\ \mathit{NPs}) \quad \text{[SRBNp2]}$$

when

$$\mathit{Int} \neq 0$$

The function add-dep2atts defined below adds its first argument as successor in its second argument. In the second condition, the first argument is made the predecessor of the second argument.

$$\frac{\begin{array}{r@{\ }l} A_1 &= \text{add-succ}(\mathsf{AP}_2,\ \mathsf{AP}_1 : \mathsf{AC}_1),\\ A_2 &= \text{add-pred}(\mathsf{AP}_1,\ \mathsf{AP}_2 : \mathsf{AC}_2) \end{array}}{\text{add-dep2atts}(\mathsf{AP}_1 : \mathsf{AC}_1,\ \mathsf{AP}_2 : \mathsf{AC}_2) = A_1;\ A_2} \quad \text{[AD2As]}$$

Below the function add-trans-deps is given. When this function is applied, the transitive dependencies are added between attributes attached to the node: *NP:NC*. *NP:ANs1;AN1;ANs2:AC1* is a parameter attribute and *NP:AN2* is a function attribute in the list of attributes. When the check whether the function attribute is a corresponding function attribute with respect to the parameter attribute (function corr-att) yields true, a dependency is added between the attributes by means of function add-corr.

$$\frac{\begin{array}{r@{\ }l} \text{true} &= \text{corr-att}(\mathsf{AN}_1,\ \mathsf{AN}_2),\\ A_1 &= \text{add-corr}(\mathsf{NP} : \mathsf{AN}_2,\ \mathsf{NP} : \mathit{ANs}_1;\ \mathsf{AN}_1;\ \mathit{ANs}_2 : \mathsf{AC}_1),\\ \mathit{As}_4 &= \text{add-trans-deps}(\mathsf{NP} : \mathsf{NC},\ \mathit{As}_1;\ \mathit{As}_2;\ \mathit{As}_3;\ A_1) \end{array}}{\begin{array}{l}\text{add-trans-deps}(\mathsf{NP} : \mathsf{NC},\ \mathit{As}_1;\ \mathsf{NP} : \mathit{ANs}_1;\ \mathsf{AN}_1;\ \mathit{ANs}_2 : \mathsf{AC}_1;\ \mathit{As}_2;\ \mathsf{NP} : \mathsf{AN}_2 : \mathsf{AC}_2;\ \mathit{As}_3) = \\ \mathit{As}_4;\ \mathsf{NP} : \mathsf{AN}_2 : \mathsf{AC}_2\end{array}} \quad \text{[ATD1]}$$

$$\frac{\begin{array}{r@{\ }l} \text{true} &= \text{corr-att}(\mathsf{AN}_1,\ \mathsf{AN}_2),\\ A_1 &= \text{add-corr}(\mathsf{NP} : \mathsf{AN}_2,\ \mathsf{NP} : \mathit{ANs}_1;\ \mathsf{AN}_1;\ \mathit{ANs}_2 : \mathsf{AC}_1),\\ \mathit{As}_4 &= \text{add-trans-deps}(\mathsf{NP} : \mathsf{NC},\ \mathit{As}_1;\ \mathit{As}_2;\ \mathit{As}_3;\ A_1) \end{array}}{\begin{array}{l}\text{add-trans-deps}(\mathsf{NP} : \mathsf{NC},\ \mathit{As}_1;\ \mathsf{NP} : \mathsf{AN}_2 : \mathsf{AC}_2;\ \mathit{As}_2;\ \mathsf{NP} : \mathit{ANs}_1;\ \mathsf{AN}_1;\ \mathit{ANs}_2 : \mathsf{AC}_1;\ \mathit{As}_3) = \\ \mathit{As}_4;\ \mathsf{NP} : \mathsf{AN}_2 : \mathsf{AC}_2\end{array}} \quad \text{[ATD2]}$$

$$\text{add-trans-deps}(N,\ \mathit{As}) \ = \ \mathit{As} \quad \textbf{otherwise} \quad \text{[ATD]}$$

The function corr-att defined below determines whether its second argument is a corresponding function attribute to its first argument by comparing their names.

$$\text{corr-att}(\mathsf{AN}_1,\ \mathsf{AN}_2) \ = \ \text{corr-att}(\texttt{"par("},\ \mathsf{AN}_1,\ \mathsf{AN}_2) \quad \text{[CA1]}$$

$$\begin{array}{l}\text{corr-att}(\text{att-name}(\texttt{"""}\ \mathit{Chars}_1\ \texttt{"""}),\\ \qquad \text{att-name}(\texttt{"""}\ \mathit{Chars}_1\ \mathit{Chars}\ \mathit{Chars}_2\ \texttt{"""}),\ \text{att-name}(\texttt{"""}\ \mathit{Chars}\ \texttt{"""})) = \text{true}\end{array} \quad \text{[CA2]}$$

$$\text{corr-att}(\mathsf{AN}_1,\ \mathsf{AN}_2,\ \mathsf{AN}_3) \ = \ \text{false} \quad \textbf{otherwise} \quad \text{[CA]}$$

H.5 SM-init-update

In the module SM-init we have defined the functions that are called out of the module Init, when the storage for a term is initialised or updated.

module SM-init-update
imports Dependencies$^{H.4}$
exports
context-free syntax

update-needed({INT ";"}*, INT, STORAGE)	→ BOOL-STORAGE
new-fun-att(ATT-NAME, STORAGE)	→ STORAGE
new-par-att({ATT-NAME ";"}+, STORAGE)	→ STORAGE
add-deps(DEPS, STORAGE)	→ STORAGE

variables

Ints [0-9′]*	→ {INT ";"}*
NodeR [0-9′]*	→ INT
PathR [0-9′]*	→ INT
PathRs [0-9′]*	→ {INT ";"}*
ChildrenRs [0-9′]*	→ {INT ";"}*

hiddens
context-free syntax

set-new-cur-init(INT, STORAGE)	→ STORAGE
new-state(INT, RANKS, {INT ";"}*)	→ STATE
present(INT, {INT ";"}*)	→ BOOL
rm-superfluous-atts(STORAGE)	→ STORAGE
rm-superf-fun-atts({ATT-PTR ";"}*, ATTRIBUTES)	→ ATTRIBUTES
rm-superf-par-atts({ATT-PTR ";"}*, ATTRIBUTES)	→ ATTRIBUTES

variables

Ints [0-9′]*	→ {INT ";"}*
NodeR [0-9′]*	→ INT
PathR [0-9′]*	→ INT
PathRs [0-9′]*	→ {INT ";"}*
ChildrenRs [0-9′]*	→ {INT ";"}*

equations

The function set-new-cur defined below sets the new current node in the storage *S*. This function set-new-cur differs from the function we the same name defined in module SM-during-reduction. If the stack in the storage is empty we are at the very begining of an initialisation, meaning that the term we are initialising the storage for is completely new, thus we will create a new storage and set the top in this just created storage as current node. The reason why we do this is that if we would chose the max of the storage as new s-top, the max in the storage would grow out of proportion if a long session takes place where constantly new terms are used. If the stack is not empty we will select the top of the stack and chose the *R*'th of its children as the new current.

$$\frac{\begin{array}{r}\text{level-of}(\text{SD}) = 0,\\ \text{path-of}(\text{SD}) = ,\\ \text{SD}_1 : Ns_1 : As_1 = \text{make-storage}(\,),\\ \text{SD}_2 = \text{set-cur}(\text{top-of}(\text{SD}_1), \text{SD}_1)\end{array}}{\text{set-new-cur-init}(R, \text{SD} : Ns : As) = \text{SD}_2 : Ns_1 : As_1} \quad \text{[SNCI1]}$$

$$\frac{\begin{array}{r}\text{level-of}(\text{SD}) = 0,\\ \text{path-of}(\text{SD}) \neq ,\\ \text{set-cur}(\text{top-of}(\text{SD}), \text{SD}) = \text{SD}_1\end{array}}{\text{set-new-cur-init}(R, \text{SD} : Ns : As) = \text{SD}_1 : Ns : As} \quad \text{[SNCI2]}$$

$$\frac{\begin{array}{l}\mathsf{level\text{-}of}(\mathsf{SD}) \neq 0,\\ \quad \mathsf{NP}_1 = \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}),\\ \quad \mathsf{NP}_2 = \mathsf{select\text{-}child}(R, \mathsf{NP}_1, Ns),\\ \quad \mathsf{SD}_1 = \mathsf{set\text{-}cur}(\mathsf{NP}_2, \mathsf{SD})\end{array}}{\mathsf{set\text{-}new\text{-}cur\text{-}init}(R, \mathsf{SD} : Ns : As) = \mathsf{SD}_1 : Ns : As} \quad [\mathsf{SNCI3}]$$

Below we define the function action. There are 5 actions possible that will result in a new state for the (new) current node in the storage:

- initialise which means that we have to (re) initialise the storage for a new (sub) term.
- update which means that a subterm of the current node has been changed and must be initialised.
- top-new-subtree which means that we have found the top of a new subterm. This means that the children of the current node must be thrown away and that dependencies with attributes in these children must be removed.
- idle means that there is no initialisation or update of the storage needed for neither the node nor its children .
- non-inc-child-changed meaning that a child of the node has been changed but this child is not the constructor of any incremental function. The only thing we have to do is to change the status of the function attributes to TE (Tobe Evaluated) and perform a prop-te-down with the function attributes.

Arguments:(The rank of the node, the path, ranks of nodes children)

$$\mathsf{new\text{-}state}(NodeR, , ChildrenRs) \ = \ \mathsf{init} \quad [\mathsf{NS1}]$$

$$\frac{\begin{array}{l}NodeR = PathR_1,\\ \mathsf{true} = \mathsf{present}(PathR_1, ChildrenRs)\end{array}}{\mathsf{new\text{-}state}(NodeR, PathR_1; PathR_2; PathRs, ChildrenRs) = \mathsf{update}} \quad [\mathsf{NS2}]$$

$$\frac{NodeR = PathR_1}{\mathsf{new\text{-}state}(NodeR, PathR_1, ChildrenRs) = \mathsf{top\text{-}new\text{-}subtree}} \quad [\mathsf{NS3}]$$

$$\frac{NodeR \neq PathR_1}{\mathsf{new\text{-}state}(NodeR, PathR_1; PathRs, ChildrenRs) = \mathsf{idle}} \quad [\mathsf{NS4}]$$

$$\frac{\begin{array}{l}NodeR = PathR_1,\\ \mathsf{false} = \mathsf{present}(PathR_2, ChildrenRs)\end{array}}{\mathsf{new\text{-}state}(NodeR, PathR_1; PathR_2; PathRs, ChildrenRs) = \mathsf{non\text{-}inc\text{-}child\text{-}changed}} \quad [\mathsf{NS5}]$$

The equations below define the function update-needed. It is a boolean function but we combine the boolean return value with a new version of the storage S enabling (simulated) side-effects. Equation $[UN1]$ below is applied if storage must be initialised for a new term and a new node is created and added to the storage. A reference to this node is pushed on the stack in the storage.

$$\frac{\begin{array}{r}\mathsf{init} = \mathsf{new\text{-}state}(R, \mathsf{path\text{-}of}(\mathsf{SD}), Rs),\\ \mathsf{SD}_1 : Ns_1 : As_1 = \mathsf{set\text{-}new\text{-}cur\text{-}init}(R, \mathsf{SD} : Ns : As),\\ N = \mathsf{make\text{-}node}(\mathsf{cur\text{-}of}(\mathsf{SD}_1), \mathsf{init}, \mathsf{max\text{-}of}(\mathsf{SD}_1), Rs),\\ \mathsf{SD}_2 = \mathsf{set\text{-}new\text{-}max}(Rs, \mathsf{SD}_1),\\ \mathsf{SD}_3 = \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_2)\end{array}}{\mathsf{update\text{-}needed}(Rs, R, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_3 : Ns_1; N : As_1} \quad [\mathsf{UN1}]$$

Equation $[UN2]$ is applied if the storage must be initialised or updated for a child of the node currently being treated. If $[UN2]$ applies we are on the right path and the state of the current node is set to update to prevent that new attributes are added later on and that no new dependencies are added. A reference to the current node in the storage is pushed on the stack and the first rank in the path is removed.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{update} & \mathsf{new\text{-}state}(R, \mathsf{path\text{-}of}(\mathsf{SD}), Rs), \\ \mathsf{SD}_1 : Ns_1 : As_1 & \mathsf{set\text{-}new\text{-}cur\text{-}init}(R, \mathsf{SD} : Ns : As), \\ N; Ns_2 & \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_1), Ns_1), \\ N_1 & \mathsf{set\text{-}state}(\mathsf{update}, N), \\ \mathsf{SD}_2 & \mathsf{remove\text{-}first\text{-}path\text{-}rank}(\mathsf{SD}_1), \\ \mathsf{SD}_3 & \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_2) \end{array}}{\mathsf{update\text{-}needed}(Rs, R, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_3 : N_1; Ns_2 : As} \quad [\mathsf{UN2}]$$

Equations $[UN3]$ is applied if the current node corresponds to the new subterm in the editor. The children of the current node are flushed and dependencies to these children are also removed by the function remove-nodes-subtrees. References to the new children are created and the state of the node is set to top-new-subtree.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{top\text{-}new\text{-}subtree} & \mathsf{new\text{-}state}(R, \mathsf{path\text{-}of}(\mathsf{SD}), Rs), \\ \mathsf{SD}_1 : Ns_1 : As_1 & \mathsf{set\text{-}new\text{-}cur\text{-}init}(R, \mathsf{SD} : Ns : As), \\ \mathsf{SD}_2 : Ns_2 : As_2 & \mathsf{remove\text{-}nodes\text{-}subtree}(\mathsf{SD}_1 : Ns_1 : As_1), \\ N; Ns_3 & \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_2), Ns_2), \\ N_1 & \mathsf{new\text{-}children}(Rs, \mathsf{max\text{-}of}(\mathsf{SD}_2), N), \\ \mathsf{SD}_3 & \mathsf{set\text{-}new\text{-}max}(Rs, \mathsf{SD}_2), \\ N_2 & \mathsf{set\text{-}state}(\mathsf{top\text{-}new\text{-}subtree}, N_1), \\ \mathsf{SD}_4 & \mathsf{push\text{-}cur\text{-}on\text{-}stack}(\mathsf{SD}_3), \\ \mathsf{SD}_5 & \mathsf{set\text{-}path}(, \mathsf{SD}_4) \end{array}}{\mathsf{update\text{-}needed}(Rs, R, \mathsf{SD} : Ns : As) = \mathsf{true} : \mathsf{SD}_5 : Ns_3; N_2 : As_2} \quad [\mathsf{UN3}]$$

Equation $[UN4]$ is applied if the path is equal to 0 which means that storage has already been (re) initialised for the new subterm in the editor or perhaps it was never changed. Another possility is that the first rank in the the path (to changed subterm) and the rank of the current node are different.

$$\frac{\mathsf{idle} = \mathsf{new\text{-}state}(R, \mathsf{path\text{-}of}(\mathsf{SD}), Rs)}{\mathsf{update\text{-}needed}(Rs, R, \mathsf{SD} : Ns : As) = \mathsf{false} : \mathsf{SD} : Ns : As} \quad [\mathsf{UN4}]$$

Equation $[UN5]$ applies if a non-incremental child of the current node in the storage has been changed. The storage contains no information concerning this child, but is aware of its existence. The status of some attributes are changed by means of function propagate-te-down and the path in the storage is set to 0; we have found the changed subterm.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{non\text{-}inc\text{-}child\text{-}changed} & \mathsf{new\text{-}state}(R, \mathsf{path\text{-}of}(\mathsf{SD}), Rs), \\ \mathsf{SD}_1 : Ns_1 : As_1 & \mathsf{set\text{-}new\text{-}cur\text{-}init}(R, \mathsf{SD} : Ns : As), \\ N; Ns_2 & \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}_1), Ns_1), \\ As_2 & \mathsf{prop\text{-}non\text{-}inc\text{-}child}(N, As_1), \\ \mathsf{SD}_2 & \mathsf{set\text{-}path}(0, \mathsf{SD}_1) \end{array}}{\mathsf{update\text{-}needed}(Rs, R, \mathsf{SD} : Ns : As) = \mathsf{false} : \mathsf{SD}_2 : Ns_1 : As_2} \quad [\mathsf{UN5}]$$

In the equations below the functions new-fun-att and new-par-att are defined. New attributes are only added to a node if the state of that node, the current node, is init. If we are dealing with a node with state update or top-new-subtree, it already has the attributes attached to it.

$$\frac{\begin{array}{r@{\;=\;}l} \mathsf{NP} : \mathsf{NC}; Ns_1 & \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}), Ns), \\ \mathsf{init} & \mathsf{state\text{-}of}(\mathsf{NP} : \mathsf{NC}), \\ A & \mathsf{make\text{-}att}(\mathsf{fun}, \mathsf{AN}, \mathsf{NP}) \end{array}}{\mathsf{new\text{-}fun\text{-}att}(\mathsf{AN}, \mathsf{SD} : Ns : As) = \mathsf{SD} : Ns : As; A} \quad [\mathsf{NFA}]$$

$$\mathsf{new\text{-}fun\text{-}att}(\mathsf{AN}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) \ = \ \mathsf{SD} : \mathit{Ns} : \mathit{As} \quad \textbf{otherwise} \qquad [\mathsf{NFA}]$$

$$\frac{\begin{aligned} \mathsf{NP} : \mathsf{NC}; \mathit{Ns}_1 &= \mathsf{find}(\mathsf{cur\text{-}of}(\mathsf{SD}), \mathit{Ns}), \\ \mathsf{init} &= \mathsf{state\text{-}of}(\mathsf{NP} : \mathsf{NC}), \\ A &= \mathsf{make\text{-}att}(\mathsf{par}, \mathit{ANs}, \mathsf{NP}) \end{aligned}}{\mathsf{new\text{-}par\text{-}att}(\mathit{ANs}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) = \mathsf{SD} : \mathit{Ns} : \mathit{As}; A} \qquad [\mathsf{NPA}]$$

$$\mathsf{new\text{-}par\text{-}att}(\mathit{ANs}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) \ = \ \mathsf{SD} : \mathit{Ns} : \mathit{As} \quad \textbf{otherwise} \qquad [\mathsf{NPA}]$$

Below the equations defining the function add-deps are given. The dependencies are only added if the state of the current node (is found on the top of the stack in the storage) is init or top-new-subtree. If the state of the current node is top-new-subtree we have to set the path to 0 after the dependencies have been added. This is to prevent that more terms are initialised.

$$\frac{\begin{aligned} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}), \\ N; \mathit{Ns}_1 &= \mathsf{find}(\mathsf{NP}, \mathit{Ns}), \\ \mathsf{init} &= \mathsf{state\text{-}of}(N), \\ \mathit{As}_1 &= \mathsf{add\text{-}new\text{-}deps}(N, \mathit{Deps}, \mathit{As}), \\ \mathit{As}_2 &= \mathsf{add\text{-}trans\text{-}deps}(N, \mathit{As}_1), \\ N_1 &= \mathsf{set\text{-}state}(\mathsf{idle}, N), \\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}), \\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP}, \mathsf{SD}_1) \end{aligned}}{\begin{array}{l} \mathsf{add\text{-}deps}(\mathit{Deps}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) = \\ \mathsf{rm\text{-}superfluous\text{-}atts}(\mathsf{SD}_2 : \mathit{Ns}_1; N_1 : \mathit{As}_2) \end{array}} \qquad [\mathsf{AD1}]$$

$$\frac{\begin{aligned} \mathsf{NP} &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}), \\ N; \mathit{Ns}_1 &= \mathsf{find}(\mathsf{NP}, \mathit{Ns}), \\ \mathsf{update} &= \mathsf{state\text{-}of}(N), \\ N_1 &= \mathsf{set\text{-}state}(\mathsf{idle}, N), \\ \mathsf{SD}_1 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}), \\ \mathsf{SD}_2 &= \mathsf{set\text{-}cur}(\mathsf{NP}, \mathsf{SD}_1) \end{aligned}}{\mathsf{add\text{-}deps}(\mathit{Deps}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) = \mathsf{SD}_2 : \mathit{Ns}_1; N_1 : \mathit{As}} \qquad [\mathsf{AD2}]$$

$$\frac{\begin{aligned} \mathsf{NP}' &= \mathsf{top\text{-}of\text{-}stack}(\mathsf{SD}), \\ \mathsf{NP} : \mathsf{NC}; \mathit{Ns}_1 &= \mathsf{find}(\mathsf{NP}', \mathit{Ns}), \\ \mathsf{top\text{-}new\text{-}subtree} &= \mathsf{state\text{-}of}(\mathsf{NP} : \mathsf{NC}), \\ \mathit{APs} &= \mathsf{fun\text{-}atts}(\mathsf{NP}, \mathit{As}), \\ \mathit{As}_1 &= \mathsf{set\text{-}status}(\mathsf{IN}, \mathit{APs}, \mathit{As}), \\ \mathit{As}_2 &= \mathsf{prop\text{-}te\text{-}up}(\mathit{APs}, \mathit{As}_1), \\ \mathit{As}_3 &= \mathsf{add\text{-}new\text{-}deps}(\mathsf{NP} : \mathsf{NC}, \mathit{Deps}, \mathit{As}_2), \\ \mathsf{SD}_1 &= \mathsf{set\text{-}path}(0, \mathsf{SD}), \\ N &= \mathsf{set\text{-}state}(\mathsf{idle}, \mathsf{NP} : \mathsf{NC}), \\ \mathsf{SD}_2 &= \mathsf{pop\text{-}of\text{-}stack}(\mathsf{SD}_1), \\ \mathsf{SD}_3 &= \mathsf{set\text{-}cur}(\mathsf{NP}, \mathsf{SD}_2) \end{aligned}}{\begin{array}{l} \mathsf{add\text{-}deps}(\mathit{Deps}, \mathsf{SD} : \mathit{Ns} : \mathit{As}) = \\ \mathsf{rm\text{-}superfluous\text{-}atts}(\mathsf{SD}_3 : \mathit{Ns}_1; N : \mathit{As}_3) \end{array}} \qquad [\mathsf{AD3}]$$

The function present defined below is an auxillary boolean function that determines if on rank is present in a list of ranks.

$$\begin{aligned} \mathsf{present}(\mathit{Int}, \mathit{Int}; \mathit{Ints}) &= \mathsf{true} && [\mathsf{Pr1}] \\ \mathsf{present}(\mathit{Int}, \mathit{Int}_1; \mathit{Ints}) &= \mathsf{present}(\mathit{Int}, \mathit{Ints}) \quad \textbf{when} \ \ \mathit{Int} \neq \mathit{Int}_1 && [\mathsf{Pr2}] \\ \mathsf{present}(\mathit{Int},) &= \mathsf{false} && [\mathsf{Pr3}] \end{aligned}$$

Below the function rm-superfluous-atts is defined. This function removes attributes related to possible injections that turned out to be no injection. We start by selecting the function attributes and parameter attributes attached to the current node and then remove these sorts of attributes separately.

$$\begin{array}{l}\text{rm-superfluous-atts}(\mathsf{SD} : Ns : As) = \\ \mathsf{SD} : Ns : As\end{array} \quad \text{[RSA]}$$

If a function attribute does not have a predecessor is removed when its successor is attached to the same node. The remove node will be a predecessor of its successor thus this reference must be removed.

$$\frac{\begin{array}{r}\mathsf{AP}_1 : \mathsf{AC}_1;\ As_1 = \text{find}(\mathsf{AP}, As), \\ \text{succs-of}(\mathsf{AP}_1 : \mathsf{AC}_1) = , \\ \text{preds-of}(\mathsf{AP}_1 : \mathsf{AC}_1) = \mathsf{NP}_1 : ANs_1, \\ \mathsf{AP} = \mathsf{NP} : ANs, \\ \mathsf{NP} = \mathsf{NP}_1, \\ A_2;\ As_2 = \text{find}(\mathsf{NP}_1 : ANs_1, As_1), \\ A_3 = \text{rm-succ}(\mathsf{AP}_1, A_2)\end{array}}{\text{rm-superf-fun-atts}(\mathsf{AP}, As) = A_3;\ As_2} \quad \text{[RSFA1]}$$

$$\frac{As_1 = \text{rm-superf-fun-atts}(\mathsf{AP}, As)}{\begin{array}{l}\text{rm-superf-fun-atts}(\mathsf{AP};\ \mathsf{AP}_1;\ APs, As) = \\ \text{rm-superf-fun-atts}(\mathsf{AP}_1;\ APs, As_1)\end{array}} \quad \text{[RSFA2]}$$

$$\text{rm-superf-fun-atts}(APs, As) \ = \ As \quad \text{[Default-RSFA]}$$

If a parameter attribute has two predcessor, the predecessor at the same node is an superfluous parameter attribute and can be removed.The reference to this predecessor must be removed.

$$\frac{\begin{array}{r}A;\ As_1 = \text{find}(\mathsf{AP}, As), \\ A = \mathsf{NP} : ANs : \mathsf{AC}_1, \\ \text{preds-of}(A) = \mathsf{AP}_1;\ \mathsf{AP}_2, \\ \mathsf{AP}_1 = \mathsf{NP}_1 : ANs_1, \\ \mathsf{NP} = \mathsf{NP}_1, \\ A_1;\ As_2 = \text{find}(\mathsf{AP}_1, As_1), \\ A_2 = \text{rm-pred}(\mathsf{AP}_1, A)\end{array}}{\text{rm-superf-par-atts}(\mathsf{AP}, As) = A_2;\ As_2} \quad \text{[RSPA1]}$$

$$\frac{\begin{array}{r}A;\ As_1 = \text{find}(\mathsf{AP}, As), \\ A = \mathsf{NP} : ANs : \mathsf{AC}_1, \\ \text{preds-of}(A) = \mathsf{AP}_1;\ \mathsf{AP}_2, \\ \mathsf{AP}_2 = \mathsf{NP}_1 : ANs_1, \\ \mathsf{NP} = \mathsf{NP}_1, \\ A_1;\ As_2 = \text{find}(\mathsf{AP}_2, As_1), \\ A_2 = \text{rm-pred}(\mathsf{AP}_1, A)\end{array}}{\text{rm-superf-par-atts}(\mathsf{AP}, As) = A_2;\ As_2} \quad \text{[RSPA2]}$$

$$\frac{As_1 = \text{rm-superf-fun-atts}(\mathsf{AP}, As)}{\begin{array}{l}\text{rm-superf-par-atts}(\mathsf{AP};\ \mathsf{AP}_1;\ APs, As) = \\ \text{rm-superf-par-atts}(\mathsf{AP}_1;\ APs, As_1)\end{array}} \quad \text{[RSPA3]}$$

$$\text{rm-superf-par-atts}(APs, As) \ = \ As \quad \text{[Default-RSPA]}$$

H.6 AuxPathSearch

module AuxPathSearch
imports InitUpdate$^{I.3}$ Integers$^{A.3}$
exports
sorts PATH PATHS TERM
context-free syntax

{INT ";"}*	→ PATH
{PATH ":"}*	→ PATHS
	→ TERM
term-top	→ INT
"path(" TERM "AND" TERM ")"	→ PATH
"path(" TERM "AND" TERM "," INT ")"	→ PATH
comp-paths({PATH ":"}*)	→ PATH
rm-term-top(PATH)	→ PATH

variables

Term [0-9′]*	→ TERM
P [0-9′]*	→ {INT ";"}*
Ps [0-9]*	→ {PATH ":"}*
R [0-9′]*	→ INT

equations

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}) = 0 \qquad \text{[BPATH1]}$$

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}') = \text{rm-term-top}(\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}', \text{term-top})) \quad \textbf{when} \quad \mathit{Term} \neq \mathit{Term}' \qquad \text{[BPATH2]}$$

$$\text{rm-term-top}(\text{term-top};\ P) = 1;\ P \qquad \text{[RM-top1]}$$

$$\text{rm-term-top}(P) = P \quad \textbf{otherwise} \qquad \text{[RM-top]}$$

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}, R) = 0 \qquad \text{[PATH1]}$$

$$\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}', \text{term-top}) = \quad \textbf{otherwise} \qquad \text{[PATH]}$$

$$\frac{R \neq \text{term-top}}{\text{path}(\ \mathit{Term}\ \text{AND}\ \mathit{Term}', R) = R} \quad \textbf{otherwise} \qquad \text{[PATH]}$$

$$\text{comp-paths}(Ps_1\text{: }0\text{: }Ps_2\text{: }P\text{: }Ps_3) = \text{comp-paths}(Ps_1\text{: }Ps_2\text{: }P\text{: }Ps_3) \qquad \text{[CompPs1]}$$

$$\text{comp-paths}(Ps_1\text{: }P\text{: }Ps_2\text{: }0\text{: }Ps_3) = \text{comp-paths}(Ps_1\text{: }P\text{: }Ps_2\text{: }Ps_3) \qquad \text{[CompPs2]}$$

$$\text{comp-paths}(P) = P \quad \textbf{when} \quad P \neq 0 \qquad \text{[CompPs3]}$$

$$\text{comp-paths}(0) = 0 \qquad \text{[CompPs4]}$$

$$\text{comp-paths}(Ps) = \quad \textbf{otherwise} \qquad \text{[CompPs]}$$

Appendix I

Generated modules

I.1 IncPico-typecheck

imports StorageInterface Booleans$^{A.2}$ Pico-syntax$^{B.5}$ Type-environments$^{B.4}$
exports
  context-free syntax

tcp(PROGRAM)	→ BOOL
tcp(PROGRAM, INCINFO)	→ BOOL
tcd(DECLS)	→ TENV
tcd(DECLS, INCINFO)	→ TENV
tcidlist(ID-TYPE-LIST, TENV)	→ TENV
tcidlist(ID-TYPE-LIST, TENV, INCINFO)	→ TENV
tcid(ID-TYPE)	→ PAIR
tcs(SERIES, TENV)	→ BOOL
tcs(SERIES, TENV, INCINFO)	→ BOOL
tcstat(STATEMENT, TENV)	→ BOOL
tcstat(STATEMENT, TENV, INCINFO)	→ BOOL
tcexpr(EXP, TENV)	→ TYPE
tcexpr(EXP, TENV, INCINFO)	→ TYPE

hiddens
  context-free syntax

nf2BOOL(NORMAL-FORM)	→ BOOL
"|" BOOL "|"	→ NORMAL-FORM
nf2TENV(NORMAL-FORM)	→ TENV
"|" TENV "|"	→ NORMAL-FORM
nf2TYPE(NORMAL-FORM)	→ TYPE
"|" TYPE "|"	→ NORMAL-FORM

  variables

BOOLvar	→ BOOL
TENVvar	→ TENV
TYPEvar	→ TYPE
PROGRAMvar	→ PROGRAM
DECLSvar	→ DECLS
ID-TYPE-LISTvar	→ ID-TYPE-LIST
SERIESvar	→ SERIES
STATEMENTvar	→ STATEMENT
EXPvar	→ EXP

equations

$$\mathrm{tcp}(\mathit{PROGRAMvar}) \;=\; \mathrm{tcp}(\mathit{PROGRAMvar}, 1:0) \qquad \text{[AII1]}$$

$$\mathrm{tcd}(\mathit{DECLSvar}) \;=\; \mathrm{tcd}(\mathit{DECLSvar}, 1:0) \qquad \text{[AII2]}$$

$$\mathrm{tcidlist}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, \mathit{Tenv}) \;=\; \mathrm{tcidlist}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, \mathit{Tenv}, 1:0) \qquad \text{[AII3]}$$

$$\mathrm{tcs}(\mathit{SERIESvar}, \mathit{Tenv}) \;=\; \mathrm{tcs}(\mathit{SERIESvar}, \mathit{Tenv}, 1:0) \qquad \text{[AII4]}$$

$$\mathrm{tcstat}(\mathit{STATEMENTvar}, \mathit{Tenv}) \;=\; \mathrm{tcstat}(\mathit{STATEMENTvar}, \mathit{Tenv}, 1:0) \qquad \text{[AII5]}$$

$$\mathrm{tcexpr}(\mathit{EXPvar}, \mathit{Tenv}) \;=\; \mathrm{tcexpr}(\mathit{EXPvar}, \mathit{Tenv}, 1:0) \qquad \text{[AII6]}$$

$$\frac{\mathrm{true} = \text{status-is-safe}(\texttt{"tcp(PROGRAM)->BOOL"}, R:L)}{\begin{array}{l}\mathrm{tcp}(\mathit{PROGRAMvar}, R:L) = \\ \mathrm{nf2BOOL}(\mathrm{retrieve}(\texttt{"tcp(PROGRAM)->BOOL"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS1]}$$

$$\frac{\mathrm{true} = \text{status-is-safe}(\texttt{"tcd(DECLS)->TENV"}, R:L)}{\mathrm{tcd}(\mathit{DECLSvar}, R:L) = \mathrm{nf2TENV}(\mathrm{retrieve}(\texttt{"tcd(DECLS)->TENV"}))} \quad \textbf{otherwise} \qquad \text{[SIS2]}$$

$$\frac{\begin{array}{l}\mathrm{true} = \text{status-is-safe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}, \\ \qquad\qquad R:L)\end{array}}{\begin{array}{l}\mathrm{tcidlist}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, \mathit{Tenv}, R:L) = \\ \mathrm{nf2TENV}(\mathrm{retrieve}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS3]}$$

$$\frac{\mathrm{true} = \text{status-is-safe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R:L)}{\begin{array}{l}\mathrm{tcs}(\mathit{SERIESvar}, \mathit{Tenv}, R:L) = \\ \mathrm{nf2BOOL}(\mathrm{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS4]}$$

$$\frac{\mathrm{true} = \text{status-is-safe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R:L)}{\begin{array}{l}\mathrm{tcstat}(\mathit{STATEMENTvar}, \mathit{Tenv}, R:L) = \\ \mathrm{nf2BOOL}(\mathrm{retrieve}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS5]}$$

$$\frac{\mathrm{true} = \text{status-is-safe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R:L)}{\begin{array}{l}\mathrm{tcexpr}(\mathit{EXPvar}, \mathit{Tenv}, R:L) = \\ \mathrm{nf2TYPE}(\mathrm{retrieve}(\texttt{"tcexpr(EXP,TENV)->TYPE"}))\end{array}} \quad \textbf{otherwise} \qquad \text{[SIS6]}$$

$$\frac{\begin{array}{l}\mathrm{true} = \text{status-is-unsafe}(\texttt{"tcp(PROGRAM)->BOOL"}, R:L), \\ \quad \mathit{Nf} = \text{store-fun-att}(\texttt{"tcp(PROGRAM)->BOOL"}, \\ \qquad\qquad\qquad | \; \mathrm{tcs}(\mathit{Series}, \mathrm{tcd}(\mathit{Decls}, 1:L+1), 2:L+1) \; |)\end{array}}{\mathrm{tcp}(\text{begin } \mathit{Decls}\ \mathit{Series} \text{ end}, R:L) = \mathrm{nf2BOOL}(\mathit{Nf})} \qquad \text{[ITc1]}$$

$$\frac{\begin{array}{l}\mathrm{true} = \text{status-is-unsafe}(\texttt{"tcd(DECLS)->TENV"}, R:L), \\ \quad \mathit{Nf} = \text{store-fun-att}(\texttt{"tcd(DECLS)->TENV"}, \\ \qquad\qquad\qquad | \; \mathrm{tcidlist}(\mathit{Id\text{-}type\text{-}list}, \text{empty-tenv}, 0:L+1) \; |)\end{array}}{\mathrm{tcd}(\text{declare } \mathit{Id\text{-}type\text{-}list};\ ,\ R:L) = \mathrm{nf2TENV}(\mathit{Nf})} \qquad \text{[ITc2]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \text{status-is-unsafe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}, R : L),\\ \mathsf{true} = \text{store-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"}, \mid Tenv \mid),\\ \quad Nf = \text{store-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\\ \qquad\qquad\qquad \mid \text{tcid}(Id\text{-}type) + Tenv \mid)\end{array}}{\text{tcidlist}(Id\text{-}type,\ Tenv,\ R : L) = \text{nf2TENV}(Nf)} \quad \text{[ITc3]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \text{status-is-unsafe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}, R : L),\\ \mathsf{true} = \text{store-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"}, \mid Tenv \mid),\\ \quad Nf = \text{store-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\\ \qquad\qquad\qquad \mid \text{tcidlist}(Id\text{-}type\text{-}list,\ \text{tcid}(Id\text{-}type) + Tenv,\ 2 : L+1) \mid)\end{array}}{\text{tcidlist}(Id\text{-}type,\ Id\text{-}type\text{-}list,\ Tenv,\ R : L) = \text{nf2TENV}(Nf)} \quad \text{[ITc4]}$$

$$\text{tcid}(Id : Type) \ = \ (Id : Type) \quad \text{[ITc5]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \text{status-is-unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L),\\ \mathsf{true} = \text{store-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, \mid Tenv \mid),\\ \quad Nf = \text{store-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\\ \qquad\qquad\qquad \mid \text{tcstat}(Stat,\ Tenv,\ 0 : L+1) \mid)\end{array}}{\text{tcs}(Stat,\ Tenv,\ R : L) = \text{nf2BOOL}(Nf)} \quad \text{[ITc6]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \text{status-is-unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L),\\ \mathsf{true} = \text{store-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, \mid Tenv \mid),\\ \quad Nf = \text{store-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\\ \qquad\qquad\qquad \mid \text{tcstat}(Stat,\ Tenv,\ 1 : L+1)\ \&\ \text{tcs}(Series,\ Tenv,\ 2 : L+1) \mid)\end{array}}{\text{tcs}(Stat;\ Series,\ Tenv,\ R : L) = \text{nf2BOOL}(Nf)} \quad \text{[ITc7]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \text{status-is-unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L),\\ \mathsf{true} = \text{store-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}, \mid Tenv \mid),\\ \quad Nf = \text{store-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\\ \qquad\qquad\qquad \mid \text{compatible}(\text{lookup } Tenv \text{ for } Id,\ \text{tcexpr}(Exp,\ Tenv,\ 2 : L+1)) \mid)\end{array}}{\text{tcstat}(Id := Exp,\ Tenv,\ R : L) = \text{nf2BOOL}(Nf)} \quad \text{[ITc8]}$$

$$\frac{\begin{array}{r@{\;}l}\mathsf{true} &= \text{status-is-unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L),\\ \mathsf{true} &= \text{store-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}, \mid Tenv \mid),\\ \mathsf{natural} &= \text{tcexpr}(Exp,\ Tenv,\ 1 : L+1),\\ Bool_1 &= \text{tcs}(Series_1,\ Tenv,\ 2 : L+1),\\ Bool_2 &= \text{tcs}(Series_2,\ Tenv,\ 3 : L+1),\\ Nf &= \text{store-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, \mid Bool_1\ \&\ Bool_2 \mid)\end{array}}{\text{tcstat}(\text{if } Exp \text{ then } Series_1 \text{ else } Series_2 \text{ fi},\ Tenv,\ R : L) = \text{nf2BOOL}(Nf)} \quad \text{[ITc9]}$$

$$\frac{\begin{array}{r@{\;}l}\mathsf{true} &= \text{status-is-unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L),\\ \mathsf{true} &= \text{store-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}, \mid Tenv \mid),\\ \mathsf{natural} &= \text{tcexpr}(Exp,\ Tenv,\ 1 : L+1),\\ Bool &= \text{tcs}(Series,\ Tenv,\ 2 : L+1),\\ Nf &= \text{store-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, \mid Bool \mid)\end{array}}{\text{tcstat}(\text{while } Exp \text{ do } Series \text{ od},\ Tenv,\ R : L) = \text{nf2BOOL}(Nf)} \quad \text{[ITc10]}$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Id}, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc11}]$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{natural} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Nat\text{-}con}, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc12}]$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{string} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Str\text{-}con}, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc13}]$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{natural} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc14}]$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1),\\ \mathsf{natural} = \mathsf{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{natural} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 - \mathit{Exp}_2, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc15}]$$

$$\frac{\begin{array}{l}\mathsf{true} = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L),\\ \mathsf{true} = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid),\\ \mathsf{string} = \mathsf{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1),\\ \mathsf{string} = \mathsf{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1),\\ \mathit{Nf} = \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{string} \mid)\end{array}}{\mathsf{tcexpr}(\mathit{Exp}_1 \parallel \mathit{Exp}_2, \mathit{Tenv}, R : L) = \mathsf{nf2TYPE}(\mathit{Nf})} \quad [\mathsf{ITc16}]$$

$$\mathsf{nf2BOOL}(\mid \mathit{BOOLvar} \mid) = \mathit{BOOLvar} \quad [\mathsf{Cv1}]$$

$$\mathsf{nf2TENV}(\mid \mathit{TENVvar} \mid) = \mathit{TENVvar} \quad [\mathsf{Cv2}]$$

$$\mathsf{nf2TYPE}(\mid \mathit{TYPEvar} \mid) = \mathit{TYPEvar} \quad [\mathsf{Cv3}]$$

I.2 IncPico-typecheckASDF

imports SM-during-reduction$^{H.2}$ Booleans$^{A.2}$ Pico-syntax$^{B.5}$ Type-environments$^{B.4}$
exports
  context-free syntax
    tcp(PROGRAM, STORAGE) → NF-STORAGE

tcp(PROGRAM, INCINFO, STORAGE) → NF-STORAGE
tcd(DECLS, STORAGE) → NF-STORAGE
tcd(DECLS, INCINFO, STORAGE) → NF-STORAGE
tcidlist(ID-TYPE-LIST, TENV, STORAGE) → NF-STORAGE
tcidlist(ID-TYPE-LIST, TENV, INCINFO, STORAGE) → NF-STORAGE
tcid(ID-TYPE) → PAIR
tcs(SERIES, TENV, STORAGE) → NF-STORAGE
tcs(SERIES, TENV, INCINFO, STORAGE) → NF-STORAGE
tcstat(STATEMENT, TENV, STORAGE) → NF-STORAGE
tcstat(STATEMENT, TENV, INCINFO, STORAGE) → NF-STORAGE
tcexpr(EXP, TENV, STORAGE) → NF-STORAGE
tcexpr(EXP, TENV, INCINFO, STORAGE) → NF-STORAGE

hiddens

context-free syntax

"|" BOOL "|" → NORMAL-FORM
"|" TENV "|" → NORMAL-FORM
"|" TYPE "|" → NORMAL-FORM

variables

BOOLvar [*0-9*]∗ → BOOL
TENVvar [*0-9*]∗ → TENV
TYPEvar [*0-9*]∗ → TYPE
PROGRAMvar → PROGRAM
DECLSvar → DECLS
ID-TYPE-LISTvar → ID-TYPE-LIST
SERIESvar → SERIES
STATEMENTvar → STATEMENT
EXPvar → EXP

equations

$$\mathrm{tcp}(\mathit{PROGRAMvar}, S) \;=\; \mathrm{tcp}(\mathit{PROGRAMvar}, 1:0, S) \qquad \text{[AII1]}$$

$$\mathrm{tcd}(\mathit{DECLSvar}, S) \;=\; \mathrm{tcd}(\mathit{DECLSvar}, 1:0, S) \qquad \text{[AII2]}$$

$$\mathrm{tcidlist}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, \mathit{Tenv}, S) = \mathrm{tcidlist}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, \mathit{Tenv}, 1:0, S) \qquad \text{[AII3]}$$

$$\mathrm{tcs}(\mathit{SERIESvar}, \mathit{Tenv}, S) \;=\; \mathrm{tcs}(\mathit{SERIESvar}, \mathit{Tenv}, 1:0, S) \qquad \text{[AII4]}$$

$$\mathrm{tcstat}(\mathit{STATEMENTvar}, \mathit{Tenv}, S) \;=\; \mathrm{tcstat}(\mathit{STATEMENTvar}, \mathit{Tenv}, 1:0, S) \qquad \text{[AII5]}$$

$$\mathrm{tcexpr}(\mathit{EXPvar}, \mathit{Tenv}, S) \;=\; \mathrm{tcexpr}(\mathit{EXPvar}, \mathit{Tenv}, 1:0, S) \qquad \text{[AII6]}$$

$$\frac{\mathrm{true} : S_1 = \mathrm{status\text{-}is\text{-}safe}(\texttt{"tcp(PROGRAM)->BOOL"}, R:L, S)}{\mathrm{tcp}(\mathit{PROGRAMvar}, R:L, S) = \mathrm{retrieve}(\texttt{"tcp(PROGRAM)->BOOL"}, S_1)} \quad \textbf{otherwise} \qquad \text{[SIS1]}$$

$$\frac{\mathrm{true} : S_1 = \mathrm{status\text{-}is\text{-}safe}(\texttt{"tcd(DECLS)->TENV"}, R:L, S)}{\mathrm{tcd}(\mathit{DECLSvar}, R:L, S) = \mathrm{retrieve}(\texttt{"tcd(DECLS)->TENV"}, S_1)} \quad \textbf{otherwise} \qquad \text{[SIS2]}$$

$$\frac{\text{true} : S_1 = \text{status-is-safe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ R : L,\ S)}{\text{tcidlist}(\textit{ID-TYPE-LISTvar},\ \textit{Tenv},\ R : L,\ S) = \text{retrieve}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ S_1)} \quad \textbf{otherwise} \qquad [\text{SIS3}]$$

$$\frac{\text{true} : S_1 = \text{status-is-safe}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ R : L,\ S)}{\text{tcs}(\textit{SERIESvar},\ \textit{Tenv},\ R : L,\ S) = \text{retrieve}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ S_1)} \quad \textbf{otherwise} \qquad [\text{SIS4}]$$

$$\frac{\text{true} : S_1 = \text{status-is-safe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\ R : L,\ S)}{\text{tcstat}(\textit{STATEMENTvar},\ \textit{Tenv},\ R : L,\ S) = \text{retrieve}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\ S_1)} \quad \textbf{otherwise} \qquad [\text{SIS5}]$$

$$\frac{\text{true} : S_1 = \text{status-is-safe}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ R : L,\ S)}{\text{tcexpr}(\textit{EXPvar},\ \textit{Tenv},\ R : L,\ S) = \text{retrieve}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\ S_1)} \quad \textbf{otherwise} \qquad [\text{SIS6}]$$

$$\frac{\begin{array}{l} \text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcp(PROGRAM)->BOOL"},\ R : L,\ S), \\ \mid \textit{TENVvar}_1 \mid : S_2 = \text{tcd}(\textit{Decls},\ 1 : L+1,\ S_1), \\ \mid \textit{BOOLvar}_1 \mid : S_3 = \text{tcs}(\textit{Series},\ \textit{TENVvar}_1,\ 2 : L+1,\ S_2) \end{array}}{\begin{array}{l} \text{tcp}(\text{begin } \textit{Decls Series} \text{ end},\ R : L,\ S) = \\ \text{store-fun-att}(\texttt{"tcp(PROGRAM)->BOOL"},\ \mid \textit{BOOLvar}_1 \mid,\ S_3) \end{array}} \qquad [\text{ITc1}]$$

$$\frac{\begin{array}{l} \text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcd(DECLS)->TENV"},\ R : L,\ S), \\ \mid \textit{TENVvar}_1 \mid : S_2 = \text{tcidlist}(\textit{Id-type-list},\ \text{empty-tenv},\ 1 : L+1,\ S_1) \end{array}}{\begin{array}{l} \text{tcd}(\text{declare } \textit{Id-type-list};\ ,\ R : L,\ S) = \\ \text{store-fun-att}(\texttt{"tcd(DECLS)->TENV"},\ \mid \textit{TENVvar}_1 \mid,\ S_2) \end{array}} \qquad [\text{ITc2}]$$

$$\frac{\begin{array}{l} \text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ R : L,\ S), \\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"},\ \mid \textit{Tenv} \mid,\ S_1) \end{array}}{\begin{array}{l} \text{tcidlist}(\textit{Id-type},\ \textit{Tenv},\ R : L,\ S) = \\ \text{store-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ \mid \text{tcid}(\textit{Id-type}) + \textit{Tenv} \mid,\ S_2) \end{array}} \qquad [\text{ITc3}]$$

$$\frac{\begin{array}{l} \text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ R : L,\ S), \\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"},\ \mid \textit{Tenv} \mid,\ S_1), \\ \mid \textit{TENVvar}_1 \mid : S_3 = \text{tcidlist}(\textit{Id-type-list},\ \text{tcid}(\textit{Id-type}) + \textit{Tenv},\ 2 : L+1,\ S_2) \end{array}}{\begin{array}{l} \text{tcidlist}(\textit{Id-type},\ \textit{Id-type-list},\ \textit{Tenv},\ R : L,\ S) = \\ \text{store-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ \mid \textit{TENVvar}_1 \mid,\ S_3) \end{array}} \qquad [\text{ITc4}]$$

$$\text{tcid}(\textit{Id} : \textit{Type}) \ = \ (\textit{Id} : \textit{Type}) \qquad [\text{ITc5}]$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, \mid \mathit{Tenv} \mid, S_1),\\ \mid \mathit{BOOLvar}_1 \mid : S_3 = \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv}, 0 : L + 1, S_2)\end{array}}{\begin{array}{l}\mathsf{tcs}(\mathit{Stat}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"}, \mid \mathit{BOOLvar}_1 \mid, S_3)\end{array}} \quad [\mathrm{ITc6}]$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcs(SERIES,TENV)->BOOL"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}, \mid \mathit{Tenv} \mid, S_1),\\ \mid \mathit{BOOLvar}_1 \mid : S_3 = \mathsf{tcstat}(\mathit{Stat}, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \mathit{BOOLvar}_2 \mid : S_4 = \mathsf{tcs}(\mathit{Series}, \mathit{Tenv}, 2 : L + 1, S_3)\end{array}}{\begin{array}{l}\mathsf{tcs}(\mathit{Stat}; \mathit{Series}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\\ \qquad \mid \mathit{BOOLvar}_1 \;\&\; \mathit{BOOLvar}_2 \mid, S_3)\end{array}} \quad [\mathrm{ITc7}]$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\\ \mid \mathit{Tenv} \mid, S_1),\\ \mid \mathit{TYPEvar}_1 \mid : S_3 = \mathsf{tcexpr}(\mathit{Exp}, \mathit{Tenv}, 2 : L + 1, S_2)\end{array}}{\begin{array}{l}\mathsf{tcstat}(\mathit{Id} := \mathit{Exp}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\\ \qquad \mid \mathsf{compatible}(\mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id}, \mathit{TYPEvar}_1) \mid, S_3)\end{array}} \quad [\mathrm{ITc8}]$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\\ \mid \mathit{Tenv} \mid, S_1),\\ \mid \mathsf{natural} \mid : S_3 = \mathsf{tcexpr}(\mathit{Exp}, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \mathit{Bool}_1 \mid : S_4 = \mathsf{tcs}(\mathit{Series}_1, \mathit{Tenv}, 2 : L + 1, S_3),\\ \mid \mathit{Bool}_2 \mid : S_5 = \mathsf{tcs}(\mathit{Series}_2, \mathit{Tenv}, 3 : L + 1, S_4)\end{array}}{\begin{array}{l}\mathsf{tcstat}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, \mid \mathit{Bool}_1 \;\&\; \mathit{Bool}_2 \mid, S_5)\end{array}} \quad [\mathrm{ITc9}]$$

$$\frac{\begin{array}{r}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\\ \mid \mathit{Tenv} \mid, S_1),\\ \mid \mathsf{natural} \mid : S_3 = \mathsf{tcexpr}(\mathit{Exp}, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \mathit{Bool} \mid : S_4 = \mathsf{tcs}(\mathit{Series}, \mathit{Tenv}, 2 : L + 1, S_3)\end{array}}{\begin{array}{l}\mathsf{tcstat}(\mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}, \mid \mathit{Bool} \mid, S_4)\end{array}} \quad [\mathrm{ITc10}]$$

$$\frac{\begin{array}{l}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid,\\ \hfill S_1)\end{array}}{\begin{array}{l}\mathsf{tcexpr}(\mathit{Id}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{lookup}\ \mathit{Tenv}\ \mathsf{for}\ \mathit{Id} \mid, S_2)\end{array}} \quad [\mathrm{ITc11}]$$

$$\frac{\begin{array}{l}\mathsf{true} : S_1 = \mathsf{status\text{-}is\text{-}unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \mathsf{true} : S_2 = \mathsf{store\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid,\\ \hfill S_1)\end{array}}{\begin{array}{l}\mathsf{tcexpr}(\mathit{Nat\text{-}con}, \mathit{Tenv}, R : L, S) =\\ \mathsf{store\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \mathsf{natural} \mid, S_2)\end{array}} \quad [\mathrm{ITc12}]$$

$$\frac{\begin{array}{l}\text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid, S_1)\end{array}}{\begin{array}{l}\text{tcexpr}(\mathit{Str\text{-}con}, \mathit{Tenv}, R : L, S) = \\ \text{store-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \text{string} \mid, S_2)\end{array}} \quad [\text{ITc13}]$$

$$\frac{\begin{array}{l}\text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid, S_1),\\ \mid \text{natural} \mid : S_3 = \text{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \text{natural} \mid : S_4 = \text{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1, S_3)\end{array}}{\begin{array}{l}\text{tcexpr}(\mathit{Exp}_1 + \mathit{Exp}_2, \mathit{Tenv}, R : L, S) = \\ \text{store-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \text{natural} \mid, S_4)\end{array}} \quad [\text{ITc14}]$$

$$\frac{\begin{array}{l}\text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid, S_1),\\ \mid \text{natural} \mid : S_3 = \text{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \text{natural} \mid : S_4 = \text{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1, S_3)\end{array}}{\begin{array}{l}\text{tcexpr}(\mathit{Exp}_1 - \mathit{Exp}_2, \mathit{Tenv}, R : L, S) = \\ \text{store-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \text{natural} \mid, S_4)\end{array}} \quad [\text{ITc15}]$$

$$\frac{\begin{array}{l}\text{true} : S_1 = \text{status-is-unsafe}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, R : L, S),\\ \text{true} : S_2 = \text{store-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}, \mid \mathit{Tenv} \mid, S_1),\\ \mid \text{string} \mid : S_3 = \text{tcexpr}(\mathit{Exp}_1, \mathit{Tenv}, 1 : L + 1, S_2),\\ \mid \text{string} \mid : S_4 = \text{tcexpr}(\mathit{Exp}_2, \mathit{Tenv}, 2 : L + 1, S_3)\end{array}}{\begin{array}{l}\text{tcexpr}(\mathit{Exp}_1 \parallel \mathit{Exp}_2, \mathit{Tenv}, R : L, S) = \\ \text{store-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}, \mid \text{string} \mid, S_4)\end{array}} \quad [\text{ITc16}]$$

I.3 InitUpdate

imports StorageInterfaceInit Pico-typecheck$^{B.6}$
exports
context-free syntax
init(PROGRAM) → BOOL
init(DECLS) → BOOL
init(ID-TYPE-LIST) → BOOL
init(SERIES) → BOOL
init(STATEMENT) → BOOL
init(EXP) → BOOL
hiddens
context-free syntax
init(PROGRAM, INT) → BOOL
init(DECLS, INT) → BOOL
init(ID-TYPE-LIST, INT) → BOOL
init(SERIES, INT) → BOOL
init(STATEMENT, INT) → BOOL
init(EXP, INT) → BOOL

variables

$PROGRAMvar \to$ PROGRAM
$DECLSvar \to$ DECLS
$ID\text{-}TYPE\text{-}LISTvar \to$ ID-TYPE-LIST
$SERIESvar \to$ SERIES
$STATEMENTvar \to$ STATEMENT
$EXPvar \to$ EXP

equations

$$\text{init}(PROGRAMvar) = \text{init}(PROGRAMvar, 1) \quad \text{[IR1]}$$

$$\text{init}(DECLSvar) = \text{init}(DECLSvar, 1) \quad \text{[IR2]}$$

$$\text{init}(ID\text{-}TYPE\text{-}LISTvar) = \text{init}(ID\text{-}TYPE\text{-}LISTvar, 1) \quad \text{[IR3]}$$

$$\text{init}(SERIESvar) = \text{init}(SERIESvar, 1) \quad \text{[IR4]}$$

$$\text{init}(STATEMENTvar) = \text{init}(STATEMENTvar, 1) \quad \text{[IR5]}$$

$$\text{init}(EXPvar) = \text{init}(EXPvar, 1) \quad \text{[IR6]}$$

$$\frac{\text{false} = \text{update-needed}(1;\, 2,\, R)}{\text{init}(\text{begin } Decls\ Series \text{ end},\, R) = \text{true}} \quad \text{[In1a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(1;\, 2,\, R),\\ Bool = \text{new-fun-att}(\texttt{"tcp(PROGRAM)->BOOL"}),\\ Bool = \text{init}(Decls, 1),\\ Bool = \text{init}(Series, 2)\end{array}}{\begin{array}{l}\text{init}(\text{begin } Decls\ Series \text{ end},\, R) =\\ \text{add-deps}(\texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \to \texttt{"tcp(PROGRAM)->BOOL"} : 0.\\ \qquad \texttt{"tcd(DECLS)->TENV"} : 1\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2)\end{array}} \quad \text{[In1b]}$$

$$\frac{\text{false} = \text{update-needed}(1,\, R)}{\text{init}(\text{declare } Id\text{-}type\text{-}list;\ ,\, R) = \text{true}} \quad \text{[In2a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(1,\, R),\\ Bool = \text{new-fun-att}(\texttt{"tcd(DECLS)->TENV"}),\\ Bool = \text{init}(Id\text{-}type\text{-}list, 1)\end{array}}{\begin{array}{l}\text{init}(\text{declare } Id\text{-}type\text{-}list;\ ,\, R) =\\ \text{add-deps}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 1\\ \qquad \to \texttt{"tcd(DECLS)->TENV"} : 0)\end{array}} \quad \text{[In2b]}$$

$$\frac{\text{false} = \text{update-needed}(0,\, R)}{\text{init}(Id\text{-}type,\, R) = \text{true}} \quad \text{[In3a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"})\end{array}}{\text{init}(\mathit{Id\text{-}type},\ R) = \text{add-deps}()} \quad \text{[In3b]}$$

$$\frac{\text{false} = \text{update-needed}(0;\ 2,\ R)}{\text{init}(\mathit{Id\text{-}type},\ \mathit{Id\text{-}type\text{-}list},\ R) = \text{true}} \quad \text{[In4a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0;\ 2,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"}),\\ \mathit{Bool} = \text{init}(\mathit{Id\text{-}type\text{-}list},\ 2)\end{array}}{\begin{array}{l}\text{init}(\mathit{Id\text{-}type},\ \mathit{Id\text{-}type\text{-}list},\ R) =\\ \text{add-deps}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 2\\ \qquad \rightarrow \texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 0)\end{array}} \quad \text{[In4b]}$$

$$\frac{\text{false} = \text{update-needed}(1;\ 2,\ R)}{\text{init}(\mathit{Stat};\ \mathit{Series},\ R) = \text{true}} \quad \text{[In5a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(1;\ 2,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}),\\ \mathit{Bool} = \text{init}(\mathit{Stat},\ 1),\\ \mathit{Bool} = \text{init}(\mathit{Series},\ 2)\end{array}}{\begin{array}{l}\text{init}(\mathit{Stat};\ \mathit{Series},\ R) =\\ \text{add-deps}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 1\\ \qquad \rightarrow \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 1.\\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \rightarrow \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2)\end{array}} \quad \text{[In5b]}$$

$$\frac{\text{false} = \text{update-needed}(0;\ 2,\ R)}{\text{init}(\mathit{Id} := \mathit{Exp},\ R) = \text{true}} \quad \text{[In6a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0;\ 2,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}),\\ \mathit{Bool} = \text{init}(\mathit{Exp},\ 2)\end{array}}{\begin{array}{l}\text{init}(\mathit{Id} := \mathit{Exp},\ R) =\\ \text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \qquad \rightarrow \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2.\\ \qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0\\ \qquad \rightarrow \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \rightarrow \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0)\end{array}} \quad \text{[In6b]}$$

$$\frac{\text{false} = \text{update-needed}(1;\, 2;\, 3,\ R)}{\text{init}(\text{if } Exp \text{ then } Series_1 \text{ else } Series_2 \text{ fi},\ R) = \text{true}} \qquad [\text{In7a}]$$

$$\frac{\begin{array}{l}
\text{true} = \text{update-needed}(1;\, 2;\, 3,\ R),\\
Bool = \text{new-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}),\\
Bool = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"}),\\
Bool = \text{new-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}),\\
Bool = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}),\\
Bool = \text{init}(Exp,\ 1),\\
Bool = \text{init}(Series_1,\ 2),\\
Bool = \text{init}(Series_2,\ 3)
\end{array}}{\begin{array}{l}
\text{init}(\text{if } Exp \text{ then } Series_1 \text{ else } Series_2 \text{ fi},\ R) =\\
\text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\
\qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\
\qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\
\qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2.\\
\qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 3\\
\qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 3.\\
\qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0\\
\qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0)
\end{array}} \qquad [\text{In7b}]$$

$$\frac{\text{false} = \text{update-needed}(1;\, 2,\ R)}{\text{init}(\text{while } Exp \text{ do } Series \text{ od},\ R) = \text{true}} \qquad [\text{In8a}]$$

$$\frac{\begin{array}{l}
\text{true} = \text{update-needed}(1;\, 2,\ R),\\
Bool = \text{new-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"}),\\
Bool = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"}),\\
Bool = \text{new-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"}),\\
Bool = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"}),\\
Bool = \text{init}(Exp,\ 1),\\
Bool = \text{init}(Series,\ 2)
\end{array}}{\begin{array}{l}
\text{init}(\text{while } Exp \text{ do } Series \text{ od},\ R) =\\
\text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\
\qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\
\qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\
\qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2.\\
\qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0\\
\qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\
\qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\
\qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0)
\end{array}} \qquad [\text{In8b}]$$

$$\frac{\text{false} = \text{update-needed}(0,\ R)}{\text{init}(\mathit{Id},\ R) = \text{true}} \quad \text{[In9a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"})\end{array}}{\text{init}(\mathit{Id},\ R) = \text{add-deps}()} \quad \text{[In9b]}$$

$$\frac{\text{false} = \text{update-needed}(0,\ R)}{\text{init}(\mathit{Nat\text{-}con},\ R) = \text{true}} \quad \text{[In10a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"})\end{array}}{\text{init}(\mathit{Nat\text{-}con},\ R) = \text{add-deps}()} \quad \text{[In10b]}$$

$$\frac{\text{false} = \text{update-needed}(0,\ R)}{\text{init}(\mathit{Str\text{-}con},\ R) = \text{true}} \quad \text{[In11a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(0,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"})\end{array}}{\text{init}(\mathit{Str\text{-}con},\ R) = \text{add-deps}()} \quad \text{[In11b]}$$

$$\frac{\text{false} = \text{update-needed}(1;\ 2,\ R)}{\text{init}(\mathit{Exp}_1\ +\ \mathit{Exp}_2,\ R) = \text{true}} \quad \text{[In12a]}$$

$$\frac{\begin{array}{l}\text{true} = \text{update-needed}(1;\ 2,\ R),\\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}),\\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}),\\ \mathit{Bool} = \text{init}(\mathit{Exp}_1,\ 1),\\ \mathit{Bool} = \text{init}(\mathit{Exp}_2,\ 2)\end{array}}{\begin{array}{l}\text{init}(\mathit{Exp}_1\ +\ \mathit{Exp}_2,\ R) =\\ \text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\ \qquad\qquad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2)\end{array}} \quad \text{[In12b]}$$

$$\frac{\text{false} = \text{update-needed}(1;\ 2,\ R)}{\text{init}(\mathit{Exp}_1\ -\ \mathit{Exp}_2,\ R) = \text{true}} \quad \text{[In13a]}$$

$$\frac{\begin{array}{l} \text{true} = \text{update-needed}(1;\, 2,\, R), \\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}), \\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}), \\ \mathit{Bool} = \text{init}(\mathit{Exp}_1,\, 1), \\ \mathit{Bool} = \text{init}(\mathit{Exp}_2,\, 2) \end{array}}{\begin{array}{l} \text{init}(\mathit{Exp}_1\; -\; \mathit{Exp}_2,\, R) = \\ \text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1 \\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0. \\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0 \\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1. \\ \qquad\qquad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2 \\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0. \\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0 \\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2) \end{array}} \quad \text{[In13b]}$$

$$\frac{\text{false} = \text{update-needed}(1;\, 2,\, R)}{\text{init}(\mathit{Exp}_1 \parallel \mathit{Exp}_2,\, R) = \text{true}} \quad \text{[In14a]}$$

$$\frac{\begin{array}{l} \text{true} = \text{update-needed}(1;\, 2,\, R), \\ \mathit{Bool} = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"}), \\ \mathit{Bool} = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"}), \\ \mathit{Bool} = \text{init}(\mathit{Exp}_1,\, 1), \\ \mathit{Bool} = \text{init}(\mathit{Exp}_2,\, 2) \end{array}}{\begin{array}{l} \text{init}(\mathit{Exp}_1 \parallel \mathit{Exp}_2,\, R) = \\ \text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1 \\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0. \\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0 \\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1. \\ \qquad\qquad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2 \\ \qquad\qquad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0. \\ \qquad\qquad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0 \\ \qquad\qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2) \end{array}} \quad \text{[In14b]}$$

$$\frac{\mathit{SERIESvar} = \mathit{STATEMENTvar}}{\text{init}(\mathit{SERIESvar},\, R) = \text{init}(\mathit{STATEMENTvar},\, R)} \quad \textbf{otherwise} \quad \text{[INJ1]}$$

I.4 InitUpdateASDF

imports SM-init-update$^{H.5}$ IncPico-typecheckASDF$^{I.2}$
exports
  context-free syntax
    init(PROGRAM, STORAGE) → STORAGE
    init(DECLS, STORAGE) → STORAGE
    init(ID-TYPE-LIST, STORAGE) → STORAGE
    init(SERIES, STORAGE) → STORAGE
    init(STATEMENT, STORAGE) → STORAGE
    init(EXP, STORAGE) → STORAGE
hiddens
  context-free syntax

$$
\begin{array}{ll}
\text{init(PROGRAM, INT, STORAGE)} & \to \text{STORAGE} \\
\text{init(DECLS, INT, STORAGE)} & \to \text{STORAGE} \\
\text{init(ID-TYPE-LIST, INT, STORAGE)} & \to \text{STORAGE} \\
\text{init(SERIES, INT, STORAGE)} & \to \text{STORAGE} \\
\text{init(STATEMENT, INT, STORAGE)} & \to \text{STORAGE} \\
\text{init(EXP, INT, STORAGE)} & \to \text{STORAGE}
\end{array}
$$

variables

$$
\begin{array}{ll}
\mathit{PROGRAMvar} & \to \text{PROGRAM} \\
\mathit{DECLSvar} & \to \text{DECLS} \\
\mathit{ID\text{-}TYPE\text{-}LISTvar} & \to \text{ID-TYPE-LIST} \\
\mathit{SERIESvar} & \to \text{SERIES} \\
\mathit{STATEMENTvar} & \to \text{STATEMENT} \\
\mathit{EXPvar} & \to \text{EXP}
\end{array}
$$

equations

$$\text{init}(\mathit{PROGRAMvar}, S_1) = \text{init}(\mathit{PROGRAMvar}, 1, S_1) \tag{IR1}$$

$$\text{init}(\mathit{DECLSvar}, S_1) = \text{init}(\mathit{DECLSvar}, 1, S_1) \tag{IR2}$$

$$\text{init}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, S_1) = \text{init}(\mathit{ID\text{-}TYPE\text{-}LISTvar}, 1, S_1) \tag{IR3}$$

$$\text{init}(\mathit{SERIESvar}, S_1) = \text{init}(\mathit{SERIESvar}, 1, S_1) \tag{IR4}$$

$$\text{init}(\mathit{STATEMENTvar}, S_1) = \text{init}(\mathit{STATEMENTvar}, 1, S_1) \tag{IR5}$$

$$\text{init}(\mathit{EXPvar}, S_1) = \text{init}(\mathit{EXPvar}, 1, S_1) \tag{IR6}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(1;\, 2,\, R,\, S_1)}{\text{init(begin } \mathit{Decls\ Series} \text{ end}, R, S_1) = S_2} \tag{In1a}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(1;\, 2,\, R,\, S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcp(PROGRAM)->BOOL"}, S_2),\\ \qquad S_4 = \text{init}(\mathit{Decls}, 1, S_3),\\ \qquad S_5 = \text{init}(\mathit{Series}, 2, S_4)\end{array}}{\begin{array}{l}\text{init(begin } \mathit{Decls\ Series} \text{ end}, R, S_1) =\\ \text{add-deps}(\texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \to \texttt{"tcp(PROGRAM)->BOOL"} : 0.\\ \qquad \texttt{"tcd(DECLS)->TENV"} : 1\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2, S_5)\end{array}} \tag{In1b}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(1,\, R,\, S_1)}{\text{init(declare } \mathit{Id\text{-}type\text{-}list};\, , R, S_1) = S_2} \tag{In2a}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(1,\, R,\, S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcd(DECLS)->TENV"}, S_2),\\ \qquad S_4 = \text{init}(\mathit{Id\text{-}type\text{-}list}, 1, S_3)\end{array}}{\begin{array}{l}\text{init(declare } \mathit{Id\text{-}type\text{-}list};\, , R, S_1) =\\ \text{add-deps}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 1\\ \qquad \to \texttt{"tcd(DECLS)->TENV"} : 0, S_4)\end{array}} \tag{In2b}$$

$$\frac{\mathsf{false} : S_2 = \text{update-needed}(0,\ R,\ S_1)}{\text{init}(\textit{Id-type},\ R,\ S_1) = S_2} \quad \text{[In3a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \text{update-needed}(0,\ R,\ S_1),\\ \quad S_3 = \text{new-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ S_2),\\ \quad S_4 = \text{new-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"},\ S_3)\end{array}}{\text{init}(\textit{Id-type},\ R,\ S_1) = \text{add-deps}(,\ S_4)} \quad \text{[In3b]}$$

$$\frac{\mathsf{false} : S_2 = \text{update-needed}(0;\ 2,\ R,\ S_1)}{\text{init}(\textit{Id-type},\ \textit{Id-type-list},\ R,\ S_1) = S_2} \quad \text{[In4a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \text{update-needed}(0;\ 2,\ R,\ S_1),\\ \quad S_3 = \text{new-fun-att}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"},\ S_2),\\ \quad S_4 = \text{new-par-att}(\texttt{"par(tcidlist(ID-TYPE-LIST,TENV)->TENV,1)"},\ S_3),\\ \quad S_5 = \text{init}(\textit{Id-type-list},\ 2,\ S_4)\end{array}}{\begin{array}{l}\text{init}(\textit{Id-type},\ \textit{Id-type-list},\ R,\ S_1) =\\ \text{add-deps}(\texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 2\\ \qquad \to \texttt{"tcidlist(ID-TYPE-LIST,TENV)->TENV"} : 0,\ S_5)\end{array}} \quad \text{[In4b]}$$

$$\frac{\mathsf{false} : S_2 = \text{update-needed}(1;\ 2,\ R,\ S_1)}{\text{init}(\textit{Stat};\ \textit{Series},\ R,\ S_1) = S_2} \quad \text{[In5a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \text{update-needed}(1;\ 2,\ R,\ S_1),\\ \quad S_3 = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\ S_2),\\ \quad S_4 = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\ S_3),\\ \quad S_5 = \text{init}(\textit{Stat},\ 1,\ S_4),\\ \quad S_6 = \text{init}(\textit{Series},\ 2,\ S_5)\end{array}}{\begin{array}{l}\text{init}(\textit{Stat};\ \textit{Series},\ R,\ S_1) =\\ \text{add-deps}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 1\\ \qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 1.\\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2,\ S_6)\end{array}} \quad \text{[In5b]}$$

$$\frac{\mathsf{false} : S_2 = \text{update-needed}(0;\ 2,\ R,\ S_1)}{\text{init}(\textit{Id} := \textit{Exp},\ R,\ S_1) = S_2} \quad \text{[In6a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(0;\, 2,\, R,\, S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\, S_2),\\ \quad S_4 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\, S_3),\\ \quad S_5 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\, S_4),\\ \quad S_6 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\, S_5),\\ \quad S_7 = \mathsf{init}(\mathit{Exp},\, 2,\, S_6)\end{array}}{\begin{array}{l}\mathsf{init}(\mathit{Id} := \mathit{Exp},\, R,\, S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2.\\ \qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0\\ \qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0,\, S_7)\end{array}} \quad \text{[In6b]}$$

$$\frac{\mathsf{false} : S_2 = \mathsf{update\text{-}needed}(1;\, 2;\, 3,\, R,\, S_1)}{\mathsf{init}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\, R,\, S_1) = S_2} \quad \text{[In7a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\, 2;\, 3,\, R,\, S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\, S_2),\\ \quad S_4 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\, S_3),\\ \quad S_5 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\, S_4),\\ \quad S_6 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\, S_5),\\ \quad S_7 = \mathsf{init}(\mathit{Exp},\, 1,\, S_6),\\ \quad S_8 = \mathsf{init}(\mathit{Series}_1,\, 2,\, S_7),\\ \quad S_9 = \mathsf{init}(\mathit{Series}_2,\, 3,\, S_8)\end{array}}{\begin{array}{l}\mathsf{init}(\mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi},\, R,\, S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2\\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2.\\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 3\\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 3.\\ \qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0\\ \qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0.\\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0\\ \qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0,\, S_9)\end{array}} \quad \text{[In7b]}$$

$$\frac{\mathsf{false} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1)}{\mathsf{init}(\mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od},\, R,\, S_1) = S_2} \quad \text{[In8a]}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(1;\,2,\,R,\,S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcstat(STATEMENT,TENV)->BOOL"},\,S_2),\\ \qquad S_4 = \text{new-fun-att}(\texttt{"tcs(SERIES,TENV)->BOOL"},\,S_3),\\ \qquad S_5 = \text{new-par-att}(\texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"},\,S_4),\\ \qquad S_6 = \text{new-par-att}(\texttt{"par(tcs(SERIES,TENV)->BOOL,1)"},\,S_5),\\ \qquad S_7 = \text{init}(\mathit{Exp},\,1,\,S_6),\\ \qquad S_8 = \text{init}(\mathit{Series},\,2,\,S_7)\end{array}}{\begin{array}{l}\text{init}(\text{while } \mathit{Exp} \text{ do } \mathit{Series} \text{ od},\,R,\,S_1) = \\ \text{add-deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1 \\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0. \\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0 \\ \qquad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1. \\ \qquad \texttt{"tcs(SERIES,TENV)->BOOL"} : 2 \\ \qquad \to \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0. \\ \qquad \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0 \\ \qquad \to \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 2. \\ \qquad \texttt{"tcstat(STATEMENT,TENV)->BOOL"} : 0 \\ \qquad \to \texttt{"tcs(SERIES,TENV)->BOOL"} : 0. \\ \qquad \texttt{"par(tcs(SERIES,TENV)->BOOL,1)"} : 0 \\ \qquad \to \texttt{"par(tcstat(STATEMENT,TENV)->BOOL,1)"} : 0,\,S_8)\end{array}} \qquad \text{[In8b]}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(0,\,R,\,S_1)}{\text{init}(\mathit{Id},\,R,\,S_1) = S_2} \qquad \text{[In9a]}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(0,\,R,\,S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\,S_2),\\ \qquad S_4 = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\,S_3)\end{array}}{\text{init}(\mathit{Id},\,R,\,S_1) = \text{add-deps}(,\,S_4)} \qquad \text{[In9b]}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(0,\,R,\,S_1)}{\text{init}(\mathit{Nat\text{-}con},\,R,\,S_1) = S_2} \qquad \text{[In10a]}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(0,\,R,\,S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\,S_2),\\ \qquad S_4 = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\,S_3)\end{array}}{\text{init}(\mathit{Nat\text{-}con},\,R,\,S_1) = \text{add-deps}(,\,S_4)} \qquad \text{[In10b]}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(0,\,R,\,S_1)}{\text{init}(\mathit{Str\text{-}con},\,R,\,S_1) = S_2} \qquad \text{[In11a]}$$

$$\frac{\begin{array}{l}\text{true} : S_2 = \text{update-needed}(0,\,R,\,S_1),\\ \qquad S_3 = \text{new-fun-att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\,S_2),\\ \qquad S_4 = \text{new-par-att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\,S_3)\end{array}}{\text{init}(\mathit{Str\text{-}con},\,R,\,S_1) = \text{add-deps}(,\,S_4)} \qquad \text{[In11b]}$$

$$\frac{\text{false} : S_2 = \text{update-needed}(1;\,2,\,R,\,S_1)}{\text{init}(\mathit{Exp}_1 + \mathit{Exp}_2,\,R,\,S_1) = S_2} \qquad \text{[In12a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\, S_2),\\ \quad S_4 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\, S_3),\\ \quad S_5 = \mathsf{init}(Exp_1,\, 1,\, S_4),\\ \quad S_6 = \mathsf{init}(Exp_2,\, 2,\, S_5)\end{array}}{\begin{array}{l}\mathsf{init}(Exp_1 + Exp_2,\, R,\, S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\ \quad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2,\, S_6)\end{array}} \quad \text{[In12b]}$$

$$\frac{\mathsf{false} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1)}{\mathsf{init}(Exp_1 - Exp_2,\, R,\, S_1) = S_2} \quad \text{[In13a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\, S_2),\\ \quad S_4 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\, S_3),\\ \quad S_5 = \mathsf{init}(Exp_1,\, 1,\, S_4),\\ \quad S_6 = \mathsf{init}(Exp_2,\, 2,\, S_5)\end{array}}{\begin{array}{l}\mathsf{init}(Exp_1 - Exp_2,\, R,\, S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\ \quad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2,\, S_6)\end{array}} \quad \text{[In13b]}$$

$$\frac{\mathsf{false} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1)}{\mathsf{init}(Exp_1 \parallel Exp_2,\, R,\, S_1) = S_2} \quad \text{[In14a]}$$

$$\frac{\begin{array}{l}\mathsf{true} : S_2 = \mathsf{update\text{-}needed}(1;\, 2,\, R,\, S_1),\\ \quad S_3 = \mathsf{new\text{-}fun\text{-}att}(\texttt{"tcexpr(EXP,TENV)->TYPE"},\, S_2),\\ \quad S_4 = \mathsf{new\text{-}par\text{-}att}(\texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"},\, S_3),\\ \quad S_5 = \mathsf{init}(Exp_1,\, 1,\, S_4),\\ \quad S_6 = \mathsf{init}(Exp_2,\, 2,\, S_5)\end{array}}{\begin{array}{l}\mathsf{init}(Exp_1 \parallel Exp_2,\, R,\, S_1) =\\ \mathsf{add\text{-}deps}(\texttt{"tcexpr(EXP,TENV)->TYPE"} : 1\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 1.\\ \quad \texttt{"tcexpr(EXP,TENV)->TYPE"} : 2\\ \quad \to \texttt{"tcexpr(EXP,TENV)->TYPE"} : 0.\\ \quad \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 0\\ \quad \to \texttt{"par(tcexpr(EXP,TENV)->TYPE,1)"} : 2,\, S_6)\end{array}} \quad \text{[In14b]}$$

$$\frac{SERIESvar = STATEMENTvar}{\text{init}(SERIESvar,\ R,\ S_1) = \text{init}(SERIESvar,\ R,\ S_1)} \quad \textbf{otherwise} \qquad \text{[INJ1]}$$

I.5 PathSearch

imports InitUpdate$^{I.3}$ AuxPathSearch$^{H.6}$
hiddens
  context-free syntax
    PROGRAM → TERM
    DECLS → TERM
    ID-TYPE-LIST → TERM
    SERIES → TERM
    EXP → TERM
  variables
    DECLSvar [*0-9*]∗ → DECLS
    ID-TYPEvar [*0-9*]∗ → ID-TYPE
    ID-TYPE-LISTvar [*0-9*]∗ → ID-TYPE-LIST
    STATEMENTvar [*0-9*]∗ → STATEMENT
    IDvar [*0-9*]∗ → ID
    SERIESvar [*0-9*]∗ → SERIES
    EXPvar [*0-9*]∗ → EXP
equations

$$\frac{\begin{array}{l} P_1 = \text{path}(\ \textit{Decls}\ \text{AND}\ \textit{DECLSvar}_1,\ 1), \\ P_2 = \text{path}(\ \textit{Series}\ \text{AND}\ \textit{SERIESvar}_1,\ 2), \\ P_3 = \text{comp-paths}(P_1\colon\ P_2), \\ P_3 \neq 0 \end{array}}{\begin{array}{l}\text{path}(\ \text{begin}\ \textit{Decls Series}\ \text{end AND begin}\ \textit{DECLSvar}_1\ \textit{SERIESvar}_1\ \text{end},\ R) = \\ R;\ P_3\end{array}} \qquad \text{[PS1]}$$

$$\frac{\begin{array}{l} P_1 = \text{path}(\ \textit{Id-type-list}\ \text{AND}\ \textit{ID-TYPE-LISTvar}_1,\ 1), \\ P_2 = \text{comp-paths}(P_1), \\ P_2 \neq 0 \end{array}}{\begin{array}{l}\text{path}(\ \text{declare}\ \textit{Id-type-list};\ \text{AND declare}\ \textit{ID-TYPE-LISTvar}_1;\ ,\ R) = \\ R;\ P_2\end{array}} \qquad \text{[PS2]}$$

$$\frac{\begin{array}{l} P_1 = \text{path}(\ \textit{Id-type}\ \text{AND}\ \textit{ID-TYPEvar}_1,\ 1), \\ P_2 = \text{path}(\ \textit{Id-type-list}\ \text{AND}\ \textit{ID-TYPE-LISTvar}_1,\ 2), \\ P_3 = \text{comp-paths}(P_1\colon\ P_2), \\ P_3 \neq 0 \end{array}}{\begin{array}{l}\text{path}(\ \textit{Id-type},\ \textit{Id-type-list}\ \text{AND}\ \textit{ID-TYPEvar}_1,\ \textit{ID-TYPE-LISTvar}_1,\ R) = \\ R;\ P_3\end{array}} \qquad \text{[PS3]}$$

$$\frac{\begin{array}{l} P_1 = \text{path}(\ \textit{Stat}\ \text{AND}\ \textit{STATEMENTvar}_1,\ 1), \\ P_2 = \text{path}(\ \textit{Series}\ \text{AND}\ \textit{SERIESvar}_1,\ 2), \\ P_3 = \text{comp-paths}(P_1\colon\ P_2), \\ P_3 \neq 0 \end{array}}{\text{path}(\ \textit{Stat};\ \textit{Series}\ \text{AND}\ \textit{STATEMENTvar}_1;\ \textit{SERIESvar}_1,\ R) = R;\ P_3} \qquad \text{[PS4]}$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Id}\ \mathsf{AND}\ \mathit{IDvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Exp}\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 2),\\ P_3 = \mathsf{comp\text{-}paths}(P_1\colon P_2),\\ P_3 \neq 0 \end{array}}{\mathsf{path}(\ \mathit{Id} := \mathit{Exp}\ \mathsf{AND}\ \mathit{IDvar}_1 := \mathit{EXPvar}_1,\ R) = R;\ P_3} \quad [\mathsf{PS5}]$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Exp}\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Series}_1\ \mathsf{AND}\ \mathit{SERIESvar}_1,\ 2),\\ P_3 = \mathsf{path}(\ \mathit{Series}_2\ \mathsf{AND}\ \mathit{SERIESvar}_2,\ 3),\\ P_4 = \mathsf{comp\text{-}paths}(P_1\colon P_2\colon P_3),\\ P_4 \neq 0 \end{array}}{\begin{array}{l} \mathsf{path}(\ \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Series}_1\ \mathsf{else}\ \mathit{Series}_2\ \mathsf{fi}\\ \mathsf{AND}\ \mathsf{if}\ \mathit{EXPvar}_1\ \mathsf{then}\ \mathit{SERIESvar}_1\ \mathsf{else}\ \mathit{SERIESvar}_2\ \mathsf{fi},\ R) =\\ R;\ P_4 \end{array}} \quad [\mathsf{PS6}]$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Exp}\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Series}\ \mathsf{AND}\ \mathit{SERIESvar}_1,\ 2),\\ P_3 = \mathsf{comp\text{-}paths}(P_1\colon P_2),\\ P_3 \neq 0 \end{array}}{\begin{array}{l} \mathsf{path}(\ \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Series}\ \mathsf{od}\ \mathsf{AND}\ \mathsf{while}\ \mathit{EXPvar}_1\ \mathsf{do}\ \mathit{SERIESvar}_1\ \mathsf{od},\ R) =\\ R;\ P_3 \end{array}} \quad [\mathsf{PS7}]$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Exp}_1\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_2,\ 2),\\ P_3 = \mathsf{comp\text{-}paths}(P_1\colon P_2),\\ P_3 \neq 0 \end{array}}{\mathsf{path}(\ \mathit{Exp}_1 + \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_1 + \mathit{EXPvar}_2,\ R) = R;\ P_3} \quad [\mathsf{PS8}]$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Exp}_1\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_2,\ 2),\\ P_3 = \mathsf{comp\text{-}paths}(P_1\colon P_2),\\ P_3 \neq 0 \end{array}}{\mathsf{path}(\ \mathit{Exp}_1 - \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_1 - \mathit{EXPvar}_2,\ R) = R;\ P_3} \quad [\mathsf{PS9}]$$

$$\frac{\begin{array}{l} P_1 = \mathsf{path}(\ \mathit{Exp}_1\ \mathsf{AND}\ \mathit{EXPvar}_1,\ 1),\\ P_2 = \mathsf{path}(\ \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_2,\ 2),\\ P_3 = \mathsf{comp\text{-}paths}(P_1\colon P_2),\\ P_3 \neq 0 \end{array}}{\mathsf{path}(\ \mathit{Exp}_1 \parallel \mathit{Exp}_2\ \mathsf{AND}\ \mathit{EXPvar}_1 \parallel \mathit{EXPvar}_2,\ R) = R;\ P_3} \quad [\mathsf{PS10}]$$