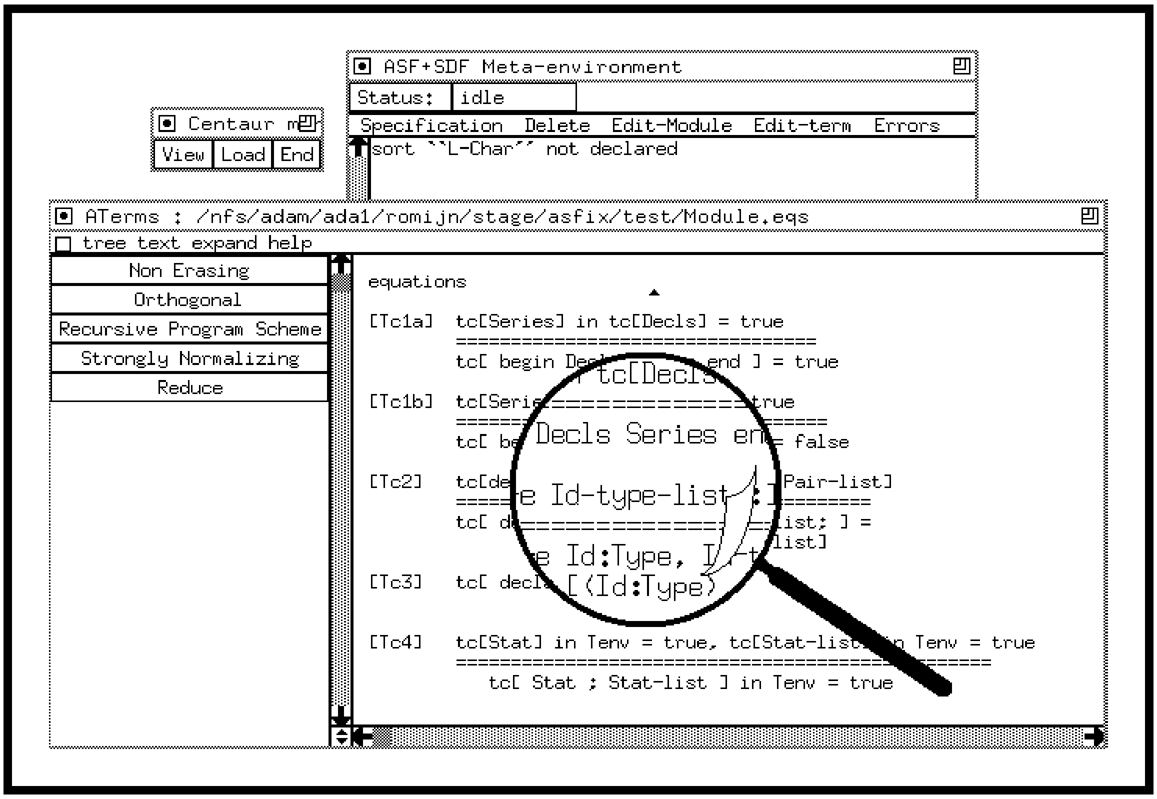
Automatic Analysis of Term Rewriting Systems

Proving properties of Term Rewriting Systems derived from ASF+SDF specifications



Judi Romijn

Automatic Analysis of Term Rewriting Systems

Proving properties of Term Rewriting Systems derived from ASF+SDF specifications

Judi Romijn

Doctoraalscriptie Informatica
Universiteit van Amsterdam
Afstudeerdocent: prof. dr. P. Klint
Examendatum: 4 juli 1995

Colofon:

Cover design by Leon Moonen

Abstract

Term rewriting systems are used frequently as an implementation technique for algebraic specifications. We present a supporting tool for the ASF+SDF Meta-environment that derives a conditional term rewriting system from an ASF+SDF specification, and tests this system for certain properties. The properties included are: *non-erasing*, *orthogonal*, *recursive program scheme* and *strongly normalizing*.

Since the specification from which we derive the conditional term rewriting system has some specific ASF+SDF features, the theory that supports this derivation is somewhat different from well-known term rewrite theories. To keep this supporting theory sound, not every ASF+SDF feature is allowed.

The property strongly normalizing is in general not decidable. We chose the method of *recursive path orderings* to be able to prove the property for a subset of the possible term rewriting systems.

The future ASF+SDF Meta-environment will be based on ASF+SDF specifications, so-called *meta-level specifications*. Although this new Meta-environment is not yet ready for use, the implementation of the tool presented here is based on the future situation.

Preface

This thesis is the result of my graduation research project carried out at the Programming Research Group of the faculty of Mathematics and Computer Science, University of Amsterdam.

I would like to thank Prof. dr. P. Klint who suggested this project and gave me the freedom to do the job as I thought best, although my plans changed quite a bit during the project.

Paul Klint, Joris Hillebrand and Wan Fokkink read draft versions of this document, showering me with useful remarks. Hans Zantema answered a very important question concerning Strong Normalization. Leon Moonen guided my first steps in LaTeX and LaTeX2$_\varepsilon$, designed the cover and provided me with neat settings, before I had even heard of or wished for them.

Finally, I have been supported by the Programming Research Group, my family and friends in various ways. I will not list names: there are too many to mention. Besides, you all know your merits.

I would like to thank you all.

Judi Romijn
Amsterdam
June 1995

Contents

Chapter 1

Introduction

1.1 The project

The ASF+SDF Meta-environment is an interactive development environment for algebraic specifications. It is being used for various purposes such as programming environment generation, code generation, etc. Debugging ASF+SDF specifications is done by parsing both syntax and equations, and reducing terms over the syntax. For such a reduction the equations are ordered from left to right and executed as term rewrite rules.

This way of debugging can be awkward. For instance, the existing implementation can trace loops in a specification only when they involve exactly one equation, and not even in all cases. An example of such an equation:

$$\mathsf{foo}(x, y) \;=\; \mathsf{foo}(1, 2) \qquad [\mathsf{Fo1}]$$

This equation will not generate an error message of any kind, but can cause your system to loop infinitely and crash. Moreover, testing a specification by reducing various terms is often an incomplete manner to prove its correctness.

For these and other reasons, we are interested in proving certain properties of ASF+SDF specifications, and we would like to be able to prove them automatically. Ideally speaking, we would want to have an ASF+SDF Meta-environment in which it is possible to push buttons for verifying properties, while building and debugging our specifications. This situation is obviously not realized yet. What we have established is the possibility to check a limited number of properties on a restricted class of specifications. To our knowledge, these are the first semantic tests in the ASF+SDF Meta-environment, apart from the reduction of terms and the Debugger.

The main restriction on the specifications is that, in most cases, they should not contain conditions or default equations. The properties are limited to the most tractable ones. As for the undecidable properties (at this point just Strong Normalization), we have chosen a strategy that either confirms the property for a specification or leaves us in doubt.

1.2 Overview of this thesis

In Chapter 2 we will describe the approach to the actual implementation by explaining the choices made at the start and the most important circumstances.

In Chapter 3 the theoretical foundation for the tests is laid. Here, the general restrictions on ASF+SDF specifications are made.

Chapter 4 to 7 give a detailed explanation of the specifications that perform the checks. Section 8 shows how the checks can be used at the moment.

Chapter 9 sums up the current results and casts a light on future extensions.

The entire ASF+SDF code is listed in the appendices.

Chapter 2

The approach

2.1 AsFix

Today, the Asf+Sdf system is still being maintained in LeLisp, but much work has been done already to create a new system in which it will be possible to maintain the system with *meta-level specifications*. This means that Asf+Sdf specifications manipulate Asf+Sdf specifications. A new fixed exchange format for specifications has been designed to make this possible [Kli94]. This format is called AsFix and is totally defined in an Asf+Sdf specification (see Appendix E).

Now, an Asf+Sdf specification is represented as an AsFix term, containing all the information in recursive or annotational form. Processing a specification is done by processing the AsFix translation of the specification. This is a term of the sort ATerm (see Appendix E).

In the light of these developments we have chosen to implement all the tests using this AsFix format.

2.2 Possible approaches

Having an Asf+Sdf specification in the AsFix format, we wanted to automatically verify certain properties. Two approaches seemed reasonable:

- Implement the property tests in meta-level Asf+Sdf specifications.
- Implement a transformation and use an external tool to do the verification.

The latter appeared to save us from inventing the wheel anew, since we could use existing expertise. However, for 'small' tests (needing only a trivial implementation), the amount of work in transforming a specification to the required input format for the external tool and getting the external tool to work on it, would by far exceed the amount of work in specifying a simple meta-level Asf+Sdf specification.

We chose to try both approaches, thinking it plausible to let meta-level Asf+Sdf specifications perform simple tests, and let the external tool perform the hard tests. We even hoped to reach such an integration level between the two, that a meta-level Asf+Sdf specification could perform a certain test, and leave hard parts of the test to the external tool.

Since an external tool existed that was even based on term rewrite theory (like the Asf+Sdf Meta-environment itself), we chose that particular tool to perform our external tests. The tool is called 'RRL', which stands for 'Rewrite Rule Laboratory'.

Unfortunately, our approach failed entirely where RRL was concerned. We never really got RRL to take the result of the transformation as legal input and we did not find a way to let RRL perform its job in the background. This failure was due to our own lack of experience in having arbitrary interactive tools work in a non-interactive way, our lack of perseverence and the bad communication with the architects of RRL, whom we asked for help by e-mail.

At this point in time, it was too late to start over by seeking another tool that might meet our needs, and we had made such progress in the meta-level ASF+SDF specifications, that we chose to concentrate entirely on that approach.

For sake of completeness, we will tell a bit about RRL in the next section. Section 2.4 explains the approach for the meta-level ASF+SDF specifications.

2.3 RRL: The Rewrite Rule Laboratory

RRL, the Rewrite Rule Laboratory, is an interactive reasoning program based on rewriting techniques ([KZ89]). It provides facilities for

- automatically proving theorems in first-order predicate calculus with equality,
- generating decision procedures for first-order (equational theories using completion procedures,
- checking the consistency and completeness of equational specifications, and
- proving theorems by induction using different approaches

For our purposes, the third option is of course the most interesting.

We can use RRL for completeness checking as follows: Start RRL in a Common Lisp interpreter (e.g. `gcl`), type `add` and enter an input specification like this:

```
[nil : term]
[list : term, term -> term]
[is-list : term -> bool]

is-list( list(X , Y)) == true
is-list( nil ) == true
]
```

The last `]` indicates the end of the input. Now type `kb` to call the Knuth-Bendix completion procedure which will try to make the specification complete in a well-known way (see [Klo92], Chapter 5). When an equation must be oriented into a rewrite rule, RRL chooses an orientation. The orientation is based on the idea RRL has of one side of the equation being smaller than the other side. If RRL cannot decide which side is smaller, it asks the user to make a choice.
We can also choose `akb` and call the automatic Knuth-Bendix completion procedure, which causes RRL to make all the choices.

Both the procedures `kb` and `akb` may not terminate.

Format problems

A lot of problems with RRL arise from the format of the input.

RRL expects variable names to start with $[x, .., z]$ and function names to start with $[a, .., w]$.

The liberal format of conditional equations cause RRL to expect keywords in strange places. If keywords appear in the wrong places, RRL either punishes the user by interpreting a meant-to-be condition as a new equation or generating an error message.

The list construction is unknown in RRL. To make RRL understand a list construction, it has to be transformed to a 'regular' function with fixed arity. For the function definition, this can be done without too many problems. Yet a general transformation of the ASF+SDF equations that preserves features like list matching is hard to implement.

Problems with conditions

RRL can handle conditions, though it is not very clear how they are interpreted. Some experiments have taught us that variable-introducing conditions can cause the completion procedure to not terminate. Yet it is in most cases obvious that by using a proper substitution to prevent the introduction of variables, we would keep the same behaviour of the specification, and have a terminating completion.

RRL is too interactive

A problem that seems easy to overcome, but actually requires a lot of UNIX and Lisp expertise, is the interactive nature of RRL. For nearly every RRL operation that is called by the user, RRL asks the user for one or more details. Since it is in general impossible to predict when RRL is going to ask for what, the only way to run RRL in the background is at this point to create a special command file, that is a concatenation of RRL commands, and that guarantees a successful termination. Such a file can only be created by experimenting with RRL manually, finding out which commands and which details are needed at a certain point, and finally putting all commands in the same order in a new file.

A solution might be to let the user interact with RRL as soon as RRL starts asking nasty questions. Still this interaction should be avoided as much as possible, for it is not desirable that an ASF+SDF Meta-environment user should learn all sorts of details about RRL.

2.4 Implementing meta-level specifications

The general idea is that we write an ASF+SDF specification, that takes a specification in the AsFix format as input, does some reducing and then tells us that the specification does or does not satisfy a certain property. For this, we translate the AsFix term that represents the specification to a new format of which the theory and layout is explained in the next chapter. The term in the new format is tested for the restrictions, posed by the test. If it succeeds, the test is performed. The result is the composition of a Boolean value, and a message indicating the success or encountered problems.

The actual and future use of the checks is explained in Section 9.1.

Chapter 3

Preliminaries

3.1 Asf+Sdf

In the Asf+Sdf Meta-environment, specifications consist of a sequence of module declarations. Each module may define syntax (Sdf) as well as semantics (Asf). In the syntax part one can prescribe the format of the terms in this syntax and define variables. In the semantics part one can specify equations, using these terms.

In order to test the specification, it is possible to form a ground term (containing no variables) in the defined syntax.

The execution of a specification is done in two steps.

First the specification is *parsed*, that is, the syntax is used to find out what the interpretation of the equations should be. If this parsing process succeeds, the specification apparently satisfies the restrictions posed by the syntax part and the general restrictions posed by the Meta-environment.

If both the specification and a term have been parsed successfully, this term can be *reduced*. This means that the equations are interpreted as (conditional) rewrite rules from left to right, so the whole specification is executed like a special kind of conditional term rewriting system. The differences with known conditional term rewriting systems are:

- Only ground terms (containing no variables) are reduced
- A leftmost-innermost reduction strategy is used while reducing a term.
- Besides normal function applications, one can use *list constructions:* (possibly empty) sequences of terms, separated by a (possibly empty) string. This separating string is called the *separator* of the list construction.
- Equations of which the left-hand side is just a variable are allowed.
- When evaluating conditions to see if a certain rewrite rule can be applied, both sides of a condition are reduced to their normal forms. The condition succeeds if the normal forms of the left-hand side and right-hand side are syntactically equal.
- Negative conditions are allowed. A negative condition succeeds if the normal forms of the left-hand side and right-hand side are syntactically not equal.
- It is possible to introduce variables in one side of conditions. The variable-introducing side is not reduced, but matched to the normal form of the other side. A positive variable-introducing condition succeeds if this matching succeeds. The variables are added to the

variables already known in this equation.
A negative variable-introducing condition always succeeds, unless the sort of the variable-introducing side has exactly one element. The ASF+SDF user manual advises against this use of negative conditions ([Kli93], Section 6.2).

- It is possible to use *default* equations. A default equation is an equation with a tag of the form default-Id. Its application is delayed until none of the 'normal' equations apply.

When we want to perform property tests on such special conditional term rewriting systems, we have to have some theoretical basis. This basis is found in the Section reftrs. We will use untyped theory to ease the comparison with existing theory. This gives a lot more possibilities than ASF+SDF would allow, but since we only use conditional term rewriting systems that we derive from 'legal' ASF+SDF specifications, we will not perform any illegal action.

In Section 3.3 we show how we actually derive a conditional term rewriting system from an ASF+SDF specification.

For a detailed description of the use of ASF+SDF, we refer to the latest User's Guide ([Kli93]) and the SDF reference manual ([HHKR92]).

3.1.1 Restrictions on ASF+SDF specifications

If we want to have a sound theoretical basis, we cannot allow all features mentioned in the previous section. We will forbid the use of

- equations of which the left-hand side is just a variable,
- negative conditions in which new variables are introduced, and
- default equations.

3.2 Term Rewriting Systems for ASF+SDF

An unconditional *Term Rewriting System* (TRS) is a pair (Σ, R) of an *alphabet* Σ and a set of rewrite rules R. The alphabet Σ consists of:

1. a countably infinite set of *variables* $\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{x}_1, \ldots : \mathcal{V}$
   and a countably infinite set *list variables* $\mathsf{xl}, \mathsf{yl}, \mathsf{zl}, \mathsf{xl}_1, \ldots : \mathcal{V}_L$
2. a non-empty set of *function symbols*. Each function symbol has either a fixed arity $n \geq 0$, or a variable arity. A function symbol with variable arity denotes a list constructor, that is, the separator of a list construction. An empty separator is denoted by "".

The set of terms over Σ is $\mathcal{T}(\Sigma)$ and is defined inductively:

1. if variable $x \in \mathcal{V}$, then $x \in \mathcal{T}(\Sigma)$,
2. if F is a function symbol with fixed arity n, $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$.
   then $\mathsf{F}(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$.
   (Note that no t_i can be a list variable)
3. if L is a function symbol with variable arity, and $t_i \in \mathcal{T}(\Sigma)$ or $t_i \in \mathcal{V}_L$ $(1 \geq i \geq n)$,
   then $\mathsf{L}(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$.
   Note that such a term is $L(\ )$ when $n = 0$.

Note that we do not allow all appearances of list variables: they can only be used within list constructions. This is conform the ASFIX notation.
Syntactic equality between terms is denoted by $\equiv$.
The set of *ground* terms over $\Sigma - (\mathcal{V} \cup \mathcal{V}_L)$ is $\mathcal{T}_0(\Sigma) \subseteq \mathcal{T}(\Sigma)$, such that any term $t \in \mathcal{T}_0(\Sigma)$ does not contain variables.

Contexts are terms containing one occurrence of a special symbol $\Box$, denoting an empty place. A context is generally denoted by $C[\;]$.
If $t \in \mathcal{T}(\Sigma)$ and t is substituted in $\Box$, the result $C[t] \in \mathcal{T}(\Sigma)$ if it is one of the inductively defined terms; t is said to be a subterm of $C[t]$.
If $C[\;]$ is not empty, t is said to be a *true* subterm of $C[t]$.

A *substitution* σ is a map from $\mathcal{T}(\Sigma) \cup \mathcal{V}_L$ to $\mathcal{T}(\Sigma)$ satisfying
$\sigma(\mathsf{F}(t_1, \ldots, t_n)) = \mathsf{F}(\sigma(t_1), \ldots, \sigma(t_n))$ for every function symbol F with fixed arity n
and $\sigma(\mathsf{L}(t_1, \ldots, t_n)) = \mathsf{L}(\sigma(t_1), \ldots, \sigma(t_n))$ for every function symbol L with variable arity. σ can map a list variable to a sequence of terms: $\sigma(\mathsf{xl}) = t_1, \ldots, t_n$ $(n \geq 0)$.
So σ is determined by its restriction to the set of variables. We also write t^σ for $\sigma(t)$.
A *ground* substitution σ is a map from $\mathcal{T}(\Sigma)$ to $\mathcal{T}_0(\Sigma)$. It leaves no variables in the term.

Each *rewrite rule*$\in R$ is a pair of terms $l, r \in \mathcal{T}(\Sigma)$, written as $l \longrightarrow r$. Two conditions will be imposed:

1. The left-hand side l is not a variable.
2. The variables in the right-hand side r are already contained in l.

A rewrite rule $l \longrightarrow r$ determines a set of *rewrites* $l^\sigma \longrightarrow r^\sigma$ for all ground substitutions σ. Now l^σ may be replaced by r^σ inside a context $C[\;]$: a *one-step rewrite* or *reduction step* $C[l^\sigma] \longrightarrow C[r^\sigma]$.

Concatenating one-step rewrites we have (possibly infinite) *reductions* $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$
If $t_0 \longrightarrow \ldots \longrightarrow t_n$ we write $t_0 \longrightarrow^* t_n$.
We say that s is in *normal form* if there is no $t \in \mathcal{T}(\Sigma)$, such that $s \longrightarrow t$.

A rule can have a label: $\mathsf{lab} : l \longrightarrow r$.
We will use both notations, assuming that $\mathsf{lab} : l \longrightarrow r$ and $l \longrightarrow r$ indicate the same rule.

Comparison between TRSs and known theory

The definitions for TRSs are restricted versions from the definitions in [Klo92], so we can use his theory.

3.2.1 Conditional Term Rewriting Systems

A conditional TRS (CTRS) is a TRS in which the rules have the following form:

$$\mathsf{lab} : \; l \longrightarrow r \text{ if } C_1 \& \cdots \& C_n \quad (C_i \;\equiv\; l_i == r_i \text{ or } C_i \;\equiv\; l_i <> r_i)$$

If we want to perform the one-step rewrite $l^\sigma \longrightarrow r^\sigma$, we will check for every condition $C_i \;\equiv\; l_i == r_i$ whether we can $\longrightarrow$-rewrite both l_i and r_i to a normal form and whether these normal forms are syntactically equal. We check for every condition $C_i \;\equiv\; l_i <> r_i$ whether we can $\longrightarrow$-rewrite both l_i and r_i to a normal form and whether these normal forms are not syntactically equal.

Note the circularity of $\longrightarrow$.

In positive conditions $C_i \equiv l_i == r_i$ it is possible to introduce new variables in l_i or r_i. The variable-introducing side is not rewritten, but matched to the normal form of the other side. The variables introduced in C_i are known in every $C_j (j > i)$ and in r. The one-step rewrite $l^\sigma \longrightarrow r^\sigma$ is performed if all conditions succeed.

Comparison between CTRSs and known theory

The definition of CTRSs is more or less conform [DOS88]'s normal join systems, which is a restriction of [Klo92]'s join systems. We will now show how any CTRS can be transformed to a normal join system without negative conditions and variable-introducing conditions, preserving the behaviour of the CTRS. From the existence of such a transformation we can conclude that our CTRS definition is a restriction of [DOS88]'s normal join systems and [Klo92]'s join systems. This enables us to use their theory for our purposes.

Negative conditions Since we rewrite each side to normal form, the test for syntactic equality is not very different from a test for syntactic inequality.
We can transform a condition $l_i <> r_i$ to $\mathsf{equal}(l_i, r_i) == \mathsf{false}$. To do this we need to know what function symbols in Σ are constructor functions. This can in most cases be determined easily.
Then we add to the CTRS:

1. definitions for the new function symbols true and false with fixed arity 0,
   the new function symbol equal with fixed arity 2
   and the new function symbol and with variable arity.
2. rules $\mathsf{and}(\mathsf{true}) \longrightarrow \mathsf{true}$
   and $\mathsf{and}(\mathsf{false}, \mathsf{xl}) \longrightarrow \mathsf{false}$
   and $\mathsf{and}(\mathsf{true}, \mathsf{x}, \mathsf{xl}) \longrightarrow \mathsf{and}(\mathsf{x}, \mathsf{xl})$
3. a rule $\mathsf{equal}(c(\mathsf{x}_1, \dots, \mathsf{x}_n), c(\mathsf{y}_1, \dots, \mathsf{y}_n)) \longrightarrow \mathsf{and}(\mathsf{equal}(\mathsf{x}_1, \mathsf{y}_1), \dots, \mathsf{equal}(\mathsf{x}_n, \mathsf{y}_n))$
   for every constructor function symbol c with fixed arity n
4. rules $\mathsf{equal}(c(\ ), c(\ )) \longrightarrow \mathsf{true}$
   and $\mathsf{equal}(c(\mathsf{x}, \mathsf{xl}), c(\mathsf{y}, \mathsf{yl})) \longrightarrow \mathsf{and}(\mathsf{equal}(\mathsf{x}_1, \mathsf{y}_1), \mathsf{equal}(c(\mathsf{xl}), c(\mathsf{yl}))$
   and $\mathsf{equal}(c(\mathsf{x}, \mathsf{xl}),\ ) \longrightarrow \mathsf{false}$
   and $\mathsf{equal}(\ , c(\mathsf{x}, \mathsf{xl})) \longrightarrow \mathsf{false}$
   for every constructor function symbol c with variable arity
5. a rule $\mathsf{equal}(c_1(\mathsf{x}_1, \dots, \mathsf{x}_n), c_2(\mathsf{y}_1, \dots, \mathsf{y}_m) \longrightarrow \mathsf{false}$
   for every two constructor function symbols c_1 and c_2 with respective fixed arities n and m, that are not syntactically equal
6. rules $\mathsf{equal}(c_1(\mathsf{x}_1, \dots, \mathsf{x}_n), c_2(\mathsf{xl})) \longrightarrow \mathsf{false}$
   and $\mathsf{equal}(c_2(\mathsf{xl}), c_1(\mathsf{x}_1, \dots, \mathsf{x}_n)) \longrightarrow \mathsf{false}$
   for constructor function symbols c_1 with fixed arity n and c_2 with variable arity
7. a rule $\mathsf{equal}(c_1(\mathsf{xl}), c_2(\mathsf{yl})) \longrightarrow \mathsf{false}$
   for every two constructor function symbols c_1 and c_2 with variable arity that are not syntactically equal

In general, all constructor functions can be found by taking all list constructors. Add all function symbols with fixed arity, for which no rule $l \longrightarrow r$ if $C_1, \dots, C_n$ can be found where l is a function application of this symbol. For a particular CTRS one can define the constructor functions oneself.

If no constructor functions exist within a CTRS, a negative condition $l_i <> r_i$ could not be evaluated successfully, and the corresponding positive condition $\mathsf{equal}(l_i, r_i) == \mathsf{false}$ would fail too.

It is not hard to see that in this manner we can get rid of the inequality without loosing any expressive power.

Variable-introducing conditions We distinguish two kinds of variable introductions:

1. $t == \mathit{Var}$. Matching Var to the normal form of t always succeeds, so we can just substitute t for every further appearance of Var and delete this condition.
   Example: $\mathsf{do\text{-}second}(x, y) \longrightarrow z$ if $\mathsf{do}(y) == z$ yields the same behaviour as $\mathsf{do\text{-}second}(x, y) \longrightarrow \mathsf{do}(y)$.

2. $t == C[\mathit{Var}]$ for some non-empty context $C[\ ]$.
   We can define a new function select with a rewrite rule $\mathsf{select}(C[\mathit{Var}]) \longrightarrow \mathit{Var}$ for this context. Now if we substitute $\mathsf{select}(t)$ for every appearance of Var including this condition, we get a new condition $t == C[\mathsf{select}(t)]$, which will succeed only if t rewrites to a normal form that can be matched with context $C[\ ]$.
   Example:
   $\mathsf{head\text{-}second}(x, y) \longrightarrow z$ if $\mathsf{head}(y) == \mathsf{concat}(\mathsf{empty}, z)$
   yields the same behaviour as
   $\mathsf{select}(\mathsf{concat}(\mathsf{empty}, z)) \longrightarrow z$
   $\mathsf{head\text{-}second}(x, y) \longrightarrow \mathsf{select}(\mathsf{head}(y))$ if $\mathsf{head}(y) == \mathsf{concat}(\mathsf{empty}, \mathsf{select}(\mathsf{head}(y))$

Note that in ASF+SDF everything is typed, so one side of a condition might be a subsort of the other side. This would make the substitutions and definition of select suggested above harder. In case *Var*'s type is a list construction, the definition of select would become impossible, since the output type of an ASF+SDF function cannot be a list construction. However, we are not considering an ASF+SDF specification, but a CTRS, in which we can define functions with these properties and still simulate the ASF+SDF behaviour. So we can ignore this restriction.

3.3 Deriving a CTRS from an ASF+SDF specification

If an ASF+SDF specification has been successfully parsed, a CTRS is *derived* from the syntax and equations of all included modules, as follows.
From the set of syntax and variable definitions, we construct the alphabet Σ:

- The lexical syntax definitions yield a function symbol with arity 0 for every lexeme.
- The context-free function definitions consist of a number of context-free elements and an output sort. The context-free elements represent the input sorts and the strings that form the name of the function. So a context-free function definition with n input sorts in the context-free elements yields a function symbol with arity n, and a name F, constructed from the strings in the context-free elements.
  Any list construction used in the context-free function definitions will yield a function symbol with variable arity and a name L which is the separator of the list construction.
- The variables are taken without alteration to form $\mathcal{V}$.

The variables and function symbols in the syntax are typed. So not all the terms in the inductively defined $\mathcal{T}(\Sigma)$ are proper ASF+SDF terms. We do not worry about this inconsistency, since the equations have to be parsed, and only contain proper ASF+SDF terms and we are not constructing new terms, except in Chapter 7, where all constructed and possibly illegal terms are intermediate.

From the set of equations we construct the set of rules R as follows:
$[<$ Tag $>]$ $C_1, ..., C_n \Longrightarrow l = r$ becomes $<$ Tag $>$: $l' \longrightarrow r'$ if C'_1 & $\cdots$ & C'_n
where

$$C_i \;\equiv\; l_i = r_i \text{ implies } C'_i \;\equiv\; l'_i == r'_i$$

and

$$C_i \;\equiv\; l_i \neq r_i \text{ implies } C'_i \;\equiv\; l'_i <> r'_i$$

Out of an ASF+SDF term t a new term t' is constructed. This is described in the following. If there are no conditions, the CTRS constructed is actually a TRS.

Lexemes

A lexeme will become a function symbol with arity 0.

Metavariables

In the ASF+SDF Meta-environment, metavariables are currently treated as constants. Every metavariable represents precisely the syntactic form it has, and is a constant of the sort it expresses. For each sort Sort we can have three metavariables: $<$ Sort $>$, $<$ Sort$*$ $>$ and $<$ Sort$+$ $>$.

Suppose you have some context-free definitions:

context-free syntax
INT$*$ $\rightarrow$ INTS
f(INTS) $\rightarrow$ INT

and equations:

f(*Int*)	=	*Int*	[F1]
f(<INT> 2)	=	2	[F2]
f(<INT$*$ > 5)	=	5	[F3]
f(*Int*$^*_1$ 3 *Int*$^*_2$)	=	f(<INT$*$ > 5)	[F4]

When you want to reduce a term f(34 2 45 5), it will surprisingly be considered to be in normal form already. However, reducing f(<INT> 2) will cause the application of Equation F2 and yield 2. Moreover, reducing f(34 3 45 5) yields a normal form 5.

In the derived CTRS, a metavariable will become a function symbol with arity 0. We will stress this fact in some of the test's explanations.

Context-free functions

A context-free function application will become a term $F(t_1, \ldots, t_n)$. Here F is the name, formed of the strings in the context-free elements, and t_i is the term, constructed out of the ith argument of the context-free function application.

List constructions

An ASF+SDF term that is a list construction will become a term $t_1\ L\ t_2\ L \cdots L\ t_n$. Here L is the separator of the list construction, and t_i is the term, constructed out of the ith argument of the list construction.

Variables

An ASF+SDF variable does not need any alteration.

Example

The following ASF+SDF equation sorts numbers in a list construction:

$$x_1 > x_2 = \mathsf{true},\ x_2 \neq 0 \Longrightarrow$$
$$\{xl_1\#x_1\#xl_2\#x_2\#xl_3\} = \{xl_1\#x_2\#xl_2\#x_1\#xl_3\}$$

It would be transformed to the rule:

$$\text{“}\{\}\text{”}(\text{“}\#\text{”}(xl_1, x_1, xl_2, x_2, xl_3)) \longrightarrow \text{“}\{\}\text{”}(\text{“}\#\text{”}(xl_1, x_2, xl_2, x_1, xl_3))$$
$$\mathsf{if}\ \ \text{“}>\text{”}(x_1, x_2) == \mathsf{true}\ \&\ x_2 <> 0$$

3.3.1 The Tree format

As mentioned before, we implemented the tests in ASF+SDF on the ASFIX representation of modules. We thought it useful to do the actual implementing on a self-designed format instead of ASFIX. Several considerations have led to this choice.

Since the start of this project, ASFIX has been under construction, and we grew tired of having to alter each meta-level function for every small change in ASFIX. The use of a self-designed format made it possible to alter only the translation function asfix2tree if ASFIX was changed syntactically. Naturally, this would not do for semantic changes.

We also wanted to state explicitly that we were performing test on the derived CTRS of a specification, meaning that the rules were oriented and interpreted in a special way.

Finally, in the light of future work on these tests, it seemed a friendly approach to try to make the self-designed format a bit easier to read than the ASFIX format. We already know about future changes to the ASF+SDF interpretations and restrictions and can never be sure if the implemented checks are completely error free.

The self-designed format is called Rules and is listed in appendix F.
The basic idea is that each symbol $\in \Sigma$ is represented as a Node of the form (Name,Arity,Sort).

Name naturally contains the name of a lexeme, function symbol, variable or metavariable. The name of a list construction is its separator.

Arity is a natural number for function symbols with fixed arity. For lexemes it is Lex, for variables it is Var, for metavariables it is MetaVar and for list constructions it is either $*$ or $+$, depending on the type of list construction.

Sort can be the name of a sort, or a list construction sort: Sortname Separator Iter, with Iter representing either $*$ or $+$.

To form a term, a Tree is constructed from these nodes that has the following form:
$\mathsf{Node}- < \mathsf{Tree\text{-}List} >$.
Tree-List contains the arguments of the Node. It is a list of Trees, separated by '|'.
If the Node is a lexeme, variable or metavariable, the Tree-List is empty.

Function applications are built as follows:
$\mathsf{fun}(t_1, \ldots, t_n)$ becomes $\mathsf{Fun}- < \mathsf{T}_1 \mid \cdots \mid \mathsf{T}_n >$
where Fun is the node constructed from fun, and T_i is the tree constructed from t_i.

List constructions are built as follows:
$\mathsf{sep}(t_1, \ldots, t_n)$ becomes $\mathsf{Sep}- < \mathsf{T}_1 \mid \mathsf{T}_2 \mid \cdots \mid \mathsf{T}_n >$
where Sep is the node constructed from sep, and T_i is the tree constructed from t_i.

Example An ASF+SDF term $\mathsf{f}(\mathit{Int}_1^*\ 3\ \mathit{Int}_2^*)$ is represented in the derived CTRS as $\mathsf{f}(\text{“”}(xl_1, 3, xl_2))$. The corresponding Tree looks like this:

$$
\begin{array}{l}
(\text{“f()”}, 1, \text{“INTS”}) - < (\text{“”}, *, \text{“INT” “” } *) - < (\text{“}\mathit{Int}_1^*\text{”}, \mathsf{Var}, \text{“INT” “” } *) - <> \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid (\text{“3”}, \mathsf{Lex}, \text{“INT”}) - <> \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid (\text{“}\mathit{Int}_2^*\text{”}, \mathsf{Var}, \text{“INT” “” } *) - <>>>
\end{array}
$$

Rules are constructed exactly like the format described in Section 3.2.1.

In none of the implemented tests we actually construct Σ, since the ASFIX terms in the equations contain all the information we needed.

Chapter 4

Checking the property *non-erasing*

4.1 The property *non-erasing*

The property *non-erasing* makes sure that no variables disappear within equations. This test is only useful in combination with the property orthogonal (see Chapter 8).

Definition 4.1 *The TRS derived from an unconditional* ASF+SDF*-specification that does not contain default-equations is* non-erasing *if in every rule $l \longrightarrow r$ the same variables occur in l and in r.*

Examples. This TRS is not non-erasing:

$$\text{"Foo1"}: \quad \mathsf{foo}(x, 1, y) \longrightarrow \mathsf{foo}(x, x, x)$$

But the following TRS is non-erasing:

$$\text{"Foo1"}: \quad \mathsf{foo}(x, 1, y) \longrightarrow \mathsf{foo}(x, x, y)$$

The definition conforms to [Klo92]. Since the property is only defined for unconditional TRSs, we report failure if we encounter a condition or a label that indicates the use of a default-equation in the ASF+SDF-specifications.

4.2 Implementation

Implementing this in ASF+SDF is not so hard. Since we expect that the specification has been parsed successfully, we only need to check for every variable in the left-hand-side whether it also appears in the right-hand-side. New variables in the right-hand-side would have led to an appropriate error message.

4.2.1 Module Non-Erasing

We need a main function non-erasing that translates AsFix to our rule-format, and that only performs the test if we are dealing with an unconditional specification.

$$\text{non-erasing}(T) \;=\; \text{non-erasing}(\text{asfix2tree}(T)) \qquad [\text{NNe1}]$$

$$\text{non-erasing}(R^*) \;=\; \text{if}(\text{check-conditions}(R^*),\, \text{ne-check}(R^*)) \qquad [\text{NNe2}]$$

Function ne-check performs the test on each rule. Function strip constructs for the left-hand-side and the right-hand-side a list of occurring variables, and these two lists are given to function compare.

$$\text{ne-check}() \;=\; \text{true} \leftrightarrow \text{Non-erasing} \qquad [\text{NNc1}]$$

$$\begin{array}{l}\text{ne-check}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^* \; R^*) = \\ \;\text{if}(\text{compare}(\text{strip}(Tr_1), \text{strip}(Tr_2)), \\ \quad \text{ne-check}(R^*), \\ \quad \text{false} \leftrightarrow \text{Erasing eq: } L)\end{array} \qquad [\text{NNc2}]$$

4.2.2 Module Compare

Function compare only needs to check for every Tree in the first list, whether it also appears in the second.

$$\begin{array}{lcl}\text{compare}(,\, Tl) & = & \text{true} \qquad [\text{Cco1}] \\ \text{compare}(N - <\; > \mid Tl_1\;,\, Tl_2) & = & \text{in}(N,\, Tl_2)\;\&\;\text{compare}(Tl_1,\, Tl_2) \qquad [\text{Cco2}]\end{array}$$

Function strip keeps all variables and discards the rest of the term. As a result, a Tree-List is constructed. Since metavariables represent syntactical constants (see Section 3.3), they are also discarded.

$$\begin{array}{lcll}\text{strip}() & = & & [\text{CSt1}] \\ \text{strip}((L,\, Nat,\, Tp) - <\, Tl\, >) & = & \text{strip}(Tl) & [\text{CSt2}] \\ \text{strip}((L,\, \text{Var},\, Tp) - <\; >) & = & (L,\, \text{Var},\, Tp) - <\; > & [\text{CSt3}] \\ \text{strip}((L,\, \text{MetaVar},\, Tp) - <\; >) & = & & [\text{CSt4}] \\ \text{strip}((L,\, \text{Lex},\, Tp) - <\, Tl\, >) & = & & [\text{CSt5}] \\ \text{strip}((L,\, *,\, Tp) - <\, Tl\, >) & = & \text{strip}(Tl) & [\text{CSt6}] \\ \text{strip}((L,\, +,\, Tp) - <\, Tl\, >) & = & \text{strip}(Tl) & [\text{CSt7}]\end{array}$$

$$\frac{\text{strip}(Tr) = Tl_1,\;\; \text{strip}(Tl^+) = Tl_2}{\text{strip}(Tr \mid Tl^+) = Tl_1 \mid Tl_2} \qquad [\text{CSt8}]$$

4.2.3 Auxiliary modules

In Appendix A the auxiliary module **NE-Booleans** is listed. Here, the composition of Boolean values and error messages is defined, as well as the necessary Boolean functions.

4.3 Extensions

Since we have no idea of the complexity of the proof for Proposition 8.6 (Chapter 8), we cannot utter a sensible word about the extension of this test for e.g. conditions.

Chapter 5

Checking the property *orthogonal*

5.1 The property *orthogonal*

The property *orthogonal* ensures that for every term it is obvious which rule should be applied, and if more rules can be applied, it does not matter in which order we apply them, since we can always reach the same term in the end. Orthogonality implies the property Church-Rosser (see Chapter 8). It can also be useful in combination with the property non-erasing.

For orthogonality we need two other property definitions.

The property *left-linear* makes sure variables appear at most once within the left-hand side of a rule.

The property *overlapping* tells us that we can form terms for which multiple rules apply, and the application of one of them can destroy the possibility of applying the others.

Definition 5.1 *An unconditional TRS is* left-linear *if in every rule $l \longrightarrow r$, l does not contain multiple occurrences of the same variable.*

Definition 5.2 *An unconditional TRS is* overlapping *if for two rules $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$, l_1 is unifiable with a non-variable subterm of l_2. This means that there is a context $Cfier[\,]$, a term t that is not a variable, and a 'most general unifier' substitution σ such that $l_2 \equiv C[t]$ and $t^\sigma \equiv {l_1}^\sigma$.*
If $l_1 \longrightarrow r_1$ is equal to $l_2 \longrightarrow r_2$, the context $C[\,]$ must be non empty.

Definition 5.3 *The CTRS derived from a conditional* ASF+SDF *specification that does not contain default equations is* orthogonal *if the unconditional version of the CTRS is left-linear and not overlapping. The unconditional version of the CTRS is obtained by simply deleting all conditions.*

Examples. The following TRS is overlapping, since we can unify "Not2" with a non-variable subterm of "Not1".

$$\begin{array}{lcl} \text{“Not1”} : & \mathsf{not}(\mathsf{not}(x)) & \longrightarrow & x \\ \text{“Not2”} : & \mathsf{not}(\mathsf{or}(x,y)) & \longrightarrow & \mathsf{and}(\mathsf{not}(x),\mathsf{not}(y)) \end{array}$$

We can unify $\mathsf{not}(\mathsf{or}(x,y))$ with $\mathsf{not}(x)$.

The following CTRS is left-linear and overlapping, so it is not orthogonal:

$$\begin{array}{lll} \text{“Foo1”} : & \mathsf{foo}(x,y,z) \longrightarrow y & \text{if} \quad x == \mathsf{true} \\ \text{“Foo2”} : & \mathsf{foo}(x,y,z) \longrightarrow z & \text{if} \quad x == \mathsf{false} \end{array}$$

It would loose the overlapping property and thus become orthogonal if we transformed it to the equivalent TRS:

$$\begin{array}{lcl} \text{“Foo1”} & : & \mathsf{foo}(\mathsf{true}, y, z) \longrightarrow y \\ \text{“Foo2”} & : & \mathsf{foo}(\mathsf{false}, y, z) \longrightarrow z \end{array}$$

The three definitions conform to [Klo92].

5.2 Implementation

Left-linearity is very easy to decide, the overlapping property takes a bit more consideration, because of the enormous amount of possibilities.

5.2.1 Module Orthogonal

The orthogonality is determined on the translation of the AsFix input, if no default equations were found.

$$\mathsf{orthogonal}(T) = \mathsf{orthogonal}(\mathsf{asfix2tree}(T)) \qquad \text{[OOr1]}$$

$$\begin{array}{l} \mathsf{orthogonal}(R^*) = \qquad \text{[OOr2]} \\ \ \mathsf{if}(\mathsf{check\text{-}default}(R^*), \\ \quad \mathsf{if}(\mathsf{left\text{-}linear}(R^*), \\ \qquad \mathsf{if}(\mathsf{invert}(\mathsf{overlapping}(R^*)), \mathsf{true} \leftrightarrow \mathsf{Left\text{-}linear\ and\ not\ overlapping}))) \end{array}$$

5.2.2 Module Left-Linear

The function left-linear performs this test on each rule. Function strip constructs for the left-hand side a list of occurring variables, and this list is given to function check-vars.

$$\mathsf{left\text{-}linear}() = \mathsf{true} \leftrightarrow \mathsf{Left\text{-}linear} \qquad \text{[LLl1]}$$

$$\begin{array}{l} \mathsf{left\text{-}linear}(L : Tr_1 \longrightarrow Tr_2\ \mathsf{if}\ C^*\ R^*) = \qquad \text{[LLl2]} \\ \ \mathsf{if}(\mathsf{check\text{-}vars}(\mathsf{strip}(Tr_1)), \mathsf{left\text{-}linear}(R^*), \mathsf{false} \leftrightarrow \mathsf{Not\ left\text{-}linear\ eq\text{:}}\ L) \end{array}$$

The function check-vars only needs to ensure that every encountered Tree does not also appear in the rest of the list.

$$\begin{array}{lcl} \mathsf{check\text{-}vars}() & = & \mathsf{true} \qquad \text{[LCv1]} \\ \mathsf{check\text{-}vars}(N - <\ >|\ Tl) & = & \mathsf{not}(\mathsf{in}(N,\ Tl))\ \&\ \mathsf{check\text{-}vars}(Tl) \qquad \text{[LCv2]} \end{array}$$

The function strip keeps all variables and discards all the rest. Metavariables are also skipped, since they do not represent a variable, but a syntactic constant.

As a result, a Tree-List is constructed.

$$\begin{array}{lcll}
\mathsf{strip}() & = & & \text{[LSt1]}\\
\mathsf{strip}((L,\, \mathit{Nat},\, \mathit{Tp}) - < \mathit{Tl} >) & = & \mathsf{strip}(\mathit{Tl}) & \text{[LSt2]}\\
\mathsf{strip}((L,\, \mathsf{Var},\, \mathit{Tp}) - < \ >) & = & (L,\, \mathsf{Var},\, \mathit{Tp}) - < \ > & \text{[LSt3]}\\
\mathsf{strip}((L,\, \mathsf{MetaVar},\, \mathit{Tp}) - < \ >) & = & & \text{[LSt4]}\\
\mathsf{strip}((L,\, \mathsf{Lex},\, \mathit{Tp}) - < \mathit{Tl} >) & = & & \text{[LSt5]}\\
\mathsf{strip}((L,\, *,\, \mathit{Tp}) - < \mathit{Tl} >) & = & \mathsf{strip}(\mathit{Tl}) & \text{[LSt6]}\\
\mathsf{strip}((L,\, +,\, \mathit{Tp}) - < \mathit{Tl} >) & = & \mathsf{strip}(\mathit{Tl}) & \text{[LSt7]}
\end{array}$$

$$\frac{\mathsf{strip}(\mathit{Tr}) = \mathit{Tl}_1,\ \ \mathsf{strip}(\mathit{Tl}^+) = \mathit{Tl}_2}{\mathsf{strip}(\mathit{Tr}|\ \mathit{Tl}^+) = \mathit{Tl}_1|\ \mathit{Tl}_2} \qquad \text{[LSt8]}$$

5.2.3 Module Overlapping

The overlapping property is determined by the function overlapping. It takes the set of rules and lets another find out whether one of the rules overlaps with another rule, and lets itself find out whether one of the rules overlaps with itself.

$$\mathsf{overlapping}(R^*) \ = \ \mathsf{another}(R^*,\, R^*) \mid \mathsf{itself}(R^*) \qquad \text{[Oov1]}$$

Since another has to let each compare a rule from the first set to every rule in the second set but not to itself, we need an auxiliary function minus that removes a particular rule from a set. The whole second set is taken along in the recursion.

$$\begin{array}{lcll}
\mathsf{another}(,\, R^*) & = & \mathsf{false} \leftrightarrow \mathsf{Not\ overlapping} & \text{[OAn1]}\\
\mathsf{another}(R\ R_1^*,\, R_2^*) & = & \mathsf{each}(R,\, R_2^*\ \mathsf{minus}\ R) \mid \mathsf{another}(R_1^*,\, R_2^*) & \text{[OAn2]}
\end{array}$$

The function each tries to unify the left-hand side of the chosen rule with the left-hand side of every rule in the set.

Note that we already know at this point that all left-hand sides contain every variable at most once!

$$\mathsf{each}(R,\,) \ = \ \mathsf{false} \leftrightarrow \mathsf{Not\ overlapping} \qquad \text{[OEa1]}$$

$$\begin{array}{ll}
\mathsf{each}(L_1 : \mathit{Tr}_1 \longrightarrow \mathit{Tr}_2\ \mathsf{if}\ C_1^*,\, L_2 : \mathit{Tr}_3 \longrightarrow \mathit{Tr}_4\ \mathsf{if}\ C_2^*\ R^*) = & \text{[OEa2]}\\
\ \mathsf{if}(\mathsf{unifiable}(\mathit{Tr}_1,\, \mathit{Tr}_3), & \\
\quad \mathsf{true} \leftrightarrow \mathsf{Overlapping\ eq{:}}\ L_1\ \mathsf{and\ eq{:}}\ L_2, & \\
\quad \mathsf{each}(L_1 : \mathit{Tr}_1 \longrightarrow \mathit{Tr}_2\ \mathsf{if}\ C_1^*,\, R^*)) &
\end{array}$$

The function itself tries to unify the left-hand side of each rule in the set with itself.

$$\mathsf{itself}() \ = \ \mathsf{false} \leftrightarrow \mathsf{Not\ overlapping} \qquad \text{[OIt1]}$$

$$\mathsf{itself}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^* \; R^*) = \qquad \text{[OIt2]}$$
$$\mathsf{if}(\mathsf{unifiable}(Tr_1),$$
$$\quad \mathsf{true} \leftrightarrow \text{Self-overlapping eq: } L,$$
$$\quad \mathsf{itself}(R^*))$$

5.2.4 Module Unifiable

In the following, the term 'fitting' is used for finding out if there exists a substitution such that the first Tree argument could become equal to the second Tree argument. When unifying, we look for the place to start fitting. When fitting, we already have bound parent Nodes to each other, and we must make the children equal. We are not trying to construct the most general unifier, we are merely showing that it can be found.
At this point we are using specific knowledge of the Trees we want to unify. Firstly, the Trees are left-linear, so no two variables need to have the same value within one Tree. Secondly, all chain functions have been preserved in the AsFix format, so there is no need to check whether we are dealing with subsorts.

Examples. In Asf+Sdf notation, g(3 *Int**) is unifiable with a subterm of f(g(*Int+*)).
In Tree notation, it looks like this:

$$\mathsf{unifiable}((\text{“g()”}, 1, \text{“INTS”}) - < (\text{“”}, +, \text{“INT” “”} +) - < (\text{“3”}, \mathsf{Lex}, \text{“INT”}) - <>$$
$$\qquad | \; (\text{“}Int^*\text{”}, \mathsf{Var}, \text{“INT” “”} *) - <>>>,$$
$$\quad (\text{“f()”}, 1, \text{“INTS”})$$
$$\quad - < (\text{“g()”}, 1, \text{“INTS”}) - < (\text{“”}, +, \text{“INT” “”} +) - < (\text{“}Int^+\text{”}, \mathsf{Var}, \text{“INT” “”} +) - <>>>>)$$

This reduces to true.

In Asf+Sdf notation, g( ) is not unifiable with a subterm of f(g(Int^+)).
In Tree notation, it looks like this:

$$\mathsf{unifiable}((\text{“g()”}, 1, \text{“INTS”}) - < (\text{“”}, +, \text{“INT” “”} +) - <>>,$$
$$\quad (\text{“f()”}, 1, \text{“INTS”})$$
$$\quad - < (\text{“g()”}, 1, \text{“INTS”}) - < (\text{“”}, +, \text{“INT” “”} +) - < (\text{“}Int^+\text{”}, \mathsf{Var}, \text{“INT” “”} +) - <>>>>)$$

This reduces to false.

In Asf+Sdf notation, not(not(*Bool*)) is unifiable with a true subterm of itself.
In Tree notation, it looks like this:

$$\mathsf{unifiable}(\text{“not()”}, 1, \text{“BOOL”}) - < (\text{“not()”}, 1, \text{“BOOL”}) - < (\text{“}Bool\text{”}, \mathsf{Var}, \text{“BOOL”}) - <>>>)$$

This reduces to true.

Function unifiable with one Tree argument will try to unify this Tree with a subterm of itself, by calling unifiable with the Tree and its list of children, to ensure that a real subterm is taken.

$$\mathsf{unifiable}(N - < Tl >) \; = \; \mathsf{unifiable}(N - < Tl > , \; Tl) \qquad \text{[UUn1]}$$

Function unifiable with two Tree arguments will try to show that the first Tree is unifiable with a subterm of the second. In case of equal roots it lays these roots onto each other and fits the lists of children. In every case it tries to unify the first Tree with one of the children of the second Tree. In this function, unifying with a variable argument is unsuccessful by definition.

There is a difference in trying to fit the children of a 'normal' function with fixed arity (fit-on), or the children of an Asf+Sdf list construction (fit-lists). In the latter case there is not just one possibility of fitting, because variables of the list sort can be unified with zero or more Trees in the other list.

$$\mathsf{unifiable}(N - < Tl > \,,\, N - < Tl >) \;=\; \mathsf{not}(\mathsf{var}(N)) \qquad \text{[UUn2]}$$

$$\frac{Tl_1 \neq Tl_2,\quad \mathsf{func}(N) = \mathsf{true}}{\mathsf{unifiable}(N - < Tl_1 > \,,\, N - < Tl_2 >) = \mathsf{fit\text{-}on}(Tl_1,\, Tl_2) \mid \mathsf{unifiable}(N - < Tl_1 > \,,\, Tl_2)} \qquad \text{[UUn3]}$$

$$\frac{Tl_1 \neq Tl_2,\quad \mathsf{list}(N) = \mathsf{true}}{\mathsf{unifiable}(N - < Tl_1 > \,,\, N - < Tl_2 >) = \mathsf{fit\text{-}lists}(Tl_1,\, Tl_2,\, \mathsf{make\text{-}type}(N)) \mid \mathsf{unifiable}(N - < Tl_1 > \,,\, Tl_2)} \qquad \text{[UUn4]}$$

$$\frac{N_1 \neq N_2,\quad \mathsf{not}(\mathsf{var}(N_1)) \;\&\; (\mathsf{func}(N_2) \mid \mathsf{list}(N_2)) = \mathsf{true}}{\mathsf{unifiable}(N_1 - < Tl_1 > \,,\, N_2 - < Tl_2 >) = \mathsf{unifiable}(N_1 - < Tl_1 > \,,\, Tl_2)} \qquad \text{[UUn5]}$$

$$\mathsf{unifiable}(Tr_1,\, Tr_2 \mid Tl^+) \;=\; \mathsf{unifiable}(Tr_1,\, Tr_2) \mid \mathsf{unifiable}(Tr_1,\, Tl^+) \qquad \text{[UUn6]}$$

$$\mathsf{unifiable}(Tr,\, Tl) \;=\; \mathsf{false} \quad \textbf{otherwise} \qquad \text{[UUn]}$$

The function fit-on tries to fit two Tree-Lists with an equal number of Trees. Here, Trees are fitted in pairs. Equal Nodes require their children lists to be fitted recursively. A variable will fit every Tree of the same type that is not a list constructor, since every list construction has a list constructor top Node that can only match an equal list constructor top Node (see Section 3.3.1). Equal Nodes require their children lists to be fitted recursively. When both lists are empty, we are done.

$$\mathsf{fit\text{-}on}(,) \;=\; \mathsf{true} \qquad \text{[UFo1]}$$

$$\frac{N_1 \neq N_2,\quad \mathsf{type}(N_1) = \mathsf{type}(N_2),\quad \mathsf{var}(N_1) \mid \mathsf{var}(N_2) = \mathsf{true}}{\mathsf{fit\text{-}on}(N_1 - < Tl_1 > \,,\, N_2 - < Tl_2 >) = \mathsf{var}(N_1) \;\&\; \mathsf{not}(\mathsf{list}(N_2)) \mid \mathsf{var}(N_2) \;\&\; \mathsf{not}(\mathsf{list}(N_1))} \qquad \text{[UFo2]}$$

$$\frac{\mathsf{list}(N) = \mathsf{false}}{\mathsf{fit\text{-}on}(N - < Tl_1 > \,,\, N - < Tl_2 >) = \mathsf{fit\text{-}on}(Tl_1,\, Tl_2)} \qquad \text{[UFo3]}$$

$$\frac{\mathsf{list}(N) = \mathsf{true}}{\mathsf{fit\text{-}on}(N - < Tl_1 > \,,\, N - < Tl_2 >) = \mathsf{fit\text{-}lists}(Tl_1,\, Tl_2,\, \mathsf{type}(N))} \qquad \text{[UFo4]}$$

$$\frac{\begin{array}{c} N_1 \neq N_2, \\ \text{list-eq}(N_1,\, N_2) = \text{true} \end{array}}{\text{fit-on}(N_1 \;-<\; Tl_1 >,\, N_2 \;-<\; Tl_2 >) = \text{fit-lists}(Tl_1,\, Tl_2,\, \text{plus-type}(N_1))} \quad \text{[UFo4]}$$

$$\text{fit-on}(Tr_1|\; Tl_1^+,\, Tr_2|\; Tl_2^+) \;=\; \text{fit-on}(Tr_1,\, Tr_2)\ \&\ \text{fit-on}(Tl_1^+,\, Tl_2^+) \quad \text{[UFo5]}$$

$$\text{fit-on}(Tl_1,\, Tl_2) \;=\; \text{false} \quad \textbf{otherwise} \quad \text{[UFo]}$$

The function fit-lists tries to fit two Tree-Lists with different numbers of Trees. This is a lot more complicated than the previous one. To simplify things a little, we keep track of the sort of list we are trying to fit in the third argument. This list type only contains a '$*$' if the type of both lists contained a '$*$'. In all other cases, at least one element has to be fitted, implying a '$+$' list type. As soon as a non-empty fitting is found, the '$+$' in the list type is transformed to '$*$' by star-type.

Examples.
fit-lists(("3", Lex, "INT")$-<>$ |("Int^*", Var, "INT" "" $*$)$-<>$,
("Int^+", Var, "INT" "" $+$)$-<>$,
"INT" "" $+$)
yields true.

fit-lists(,
("Int^+", Var, "INT" "" $+$)$-<>$,
"INT" "" $+$)
yields false.

Auxiliary functions.
Function is-elt succeeds if the type of the Node is an element of the list type.
Function list-eq succeeds if two types represent the same list type, despite '$*$'/'$+$' differences.
Function star succeeds if a type is a list type with '$*$'.

The empty lists only fit if the list type contains a '$*$'.

$$\text{fit-lists}(,\, ,\, Tp) \;=\; \text{star}(Tp) \quad \text{[UFl1]}$$

Variables of the '$*$' list type are allowed to fit an empty list.

$$\frac{\begin{array}{c} \text{var}(N)\ \&\ \text{star}(Tp) = \text{true}, \\ \text{type}(N) = Tp \end{array}}{\text{fit-lists}(,\, N -<\;>|\; Tl_2\, ,\, Tp) = \text{fit-lists}(,\, Tl_2,\, Tp)} \quad \text{[UFl2]}$$

$$\frac{\begin{array}{c} \text{var}(N)\ \&\ \text{star}(Tp) = \text{true}, \\ \text{type}(N) = Tp \end{array}}{\text{fit-lists}(N -<\;>|\; Tl_2\, ,\, ,\, Tp) = \text{fit-lists}(Tl_2,\, ,\, Tp)} \quad \text{[UFl3]}$$

Elements of the list type can only fit each other.

$$\frac{\textsf{is-elt}(N_1,\ Tp)\ \&\ \textsf{is-elt}(N_2,\ Tp) = \textsf{true}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < Tl_1 >|\ Tl_2\ ,\ N_2 - < Tl_3 >|\ Tl_4\ ,\ Tp) = \\ \textsf{fit-on}(N_1 - < Tl_1 >\ ,\ N_2 - < Tl_3 >)\ \&\ \textsf{fit-lists}(Tl_2,\ Tl_4,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl4}]$$

Non-variable Nodes of the list type (be it '$*$' or '$+$') can only fit each other.

$$\frac{\begin{array}{c}\textsf{not}(\textsf{var}(N_1))\ \&\ \textsf{not}(\textsf{var}(N_2)) = \textsf{true}, \\ \textsf{list-eq}(\textsf{type}(N_1),\ Tp)\ \&\ \textsf{list-eq}(\textsf{type}(N_2),\ Tp) = \textsf{true}\end{array}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < Tl_1 >|\ Tl_2\ ,\ N_2 - < Tl_3 >|\ Tl_4\ ,\ Tp) = \\ \textsf{fit-on}(N_1 - < Tl_1 >\ ,\ N_2 - < Tl_3 >)\ \&\ \textsf{fit-lists}(Tl_2,\ Tl_4,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl5}]$$

One top Node is a variable of the '$*$' list type, the other an element or a non-variable Node of the list type. The variable can fit nothing, or just this Node, or this Node and more of the list.

$$\frac{\begin{array}{c}\textsf{var}(N_2)\ \&\ \textsf{list-eq}(\textsf{type}(N_2),\ Tp)\ \&\ \textsf{star}(\textsf{type}(N_2)) = \textsf{true}, \\ \textsf{is-elt}(N_1,\ Tp)\ |\ \textsf{list-eq}(\textsf{type}(N_1),\ Tp)\ \&\ \textsf{not}(\textsf{var}(N_1)) = \textsf{true}\end{array}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < Tl_1 >|\ Tl_2\ ,\ N_2 - < >|\ Tl_3\ ,\ Tp) = \\ \textsf{fit-lists}(N_1 - < Tl_1 >|\ Tl_2\ ,\ Tl_3,\ Tp) \\ |\ \textsf{fit-lists}(Tl_2,\ Tl_3,\ \textsf{star-type}(Tp)) \\ |\ \textsf{fit-lists}(Tl_2,\ N_2 - < >|\ Tl_3\ ,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl6}]$$

$$\frac{\begin{array}{c}\textsf{var}(N_1)\ \&\ \textsf{list-eq}(\textsf{type}(N_1),\ Tp)\ \&\ \textsf{star}(\textsf{type}(N_1)) = \textsf{true}, \\ \textsf{is-elt}(N_2,\ Tp)\ |\ \textsf{list-eq}(\textsf{type}(N_2),\ Tp)\ \&\ \textsf{not}(\textsf{var}(N_2)) = \textsf{true}\end{array}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < >|\ Tl_1\ ,\ N_2 - < Tl_2 >|\ Tl_3\ ,\ Tp) = \\ \textsf{fit-lists}(Tl_1,\ N_2 - < Tl_2 >|\ Tl_3\ ,\ Tp) \\ |\ \textsf{fit-lists}(Tl_1,\ Tl_3,\ \textsf{star-type}(Tp)) \\ |\ \textsf{fit-lists}(N_1 - < >|\ Tl_1\ ,\ Tl_3,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl7}]$$

One top Node is a variable of the '$+$' list type, the other an element or a non-variable Node of the list type. The variable can fit just this Node, or this Node and more of the list.

$$\frac{\begin{array}{c}\textsf{var}(N_2)\ \&\ \textsf{list-eq}(\textsf{type}(N_2),\ Tp)\ \&\ \textsf{not}(\textsf{star}(\textsf{type}(N_2))) = \textsf{true}, \\ \textsf{is-elt}(N_1,\ Tp)\ |\ \textsf{list-eq}(\textsf{type}(N_1),\ Tp)\ \&\ \textsf{not}(\textsf{var}(N_1)) = \textsf{true}\end{array}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < Tl_1 >|\ Tl_2\ ,\ N_2 - < >|\ Tl_3\ ,\ Tp) = \\ \textsf{fit-lists}(Tl_2,\ Tl_3,\ \textsf{star-type}(Tp)) \\ |\ \textsf{fit-lists}(Tl_2,\ \textsf{star-type}(N_2) - < >|\ Tl_3\ ,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl8}]$$

$$\frac{\begin{array}{c}\textsf{var}(N_1)\ \&\ \textsf{list-eq}(\textsf{type}(N_1),\ Tp)\ \&\ \textsf{not}(\textsf{star}(\textsf{type}(N_1))) = \textsf{true}, \\ \textsf{is-elt}(N_2,\ Tp)\ |\ \textsf{list-eq}(\textsf{type}(N_2),\ Tp)\ \&\ \textsf{not}(\textsf{var}(N_2)) = \textsf{true}\end{array}}{\begin{array}{l}\textsf{fit-lists}(N_1 - < >|\ Tl_1\ ,\ N_2 - < Tl_2 >|\ Tl_3\ ,\ Tp) = \\ \textsf{fit-lists}(Tl_1,\ Tl_3,\ \textsf{star-type}(Tp)) \\ |\ \textsf{fit-lists}(\textsf{star-type}(N_1) - < >|\ Tl_1\ ,\ Tl_3,\ \textsf{star-type}(Tp))\end{array}} \quad [\textsf{UFl9}]$$

Two variables of the list type can fit each other exactly, or one of them fits the other and more of the list.

$$\frac{\begin{array}{r}\mathsf{var}(N_1)\ \&\ \mathsf{var}(N_2) = \mathsf{true},\\ \mathsf{list\text{-}eq}(\mathsf{type}(N_1),\ Tp)\ \&\ \mathsf{list\text{-}eq}(\mathsf{type}(N_2),\ Tp) = \mathsf{true}\end{array}}{\begin{array}{l}\mathsf{fit\text{-}lists}(N_1 - <\ >|\ Tl_1\ ,\ N_2 - <\ >|\ Tl_2\ ,\ Tp) =\\ \mathsf{fit\text{-}lists}(Tl_1,\ Tl_2,\ \mathsf{star\text{-}type}(Tp))\\ |\ \mathsf{fit\text{-}lists}(\mathsf{star\text{-}type}(N_1) - <\ >|\ Tl_1\ ,\ Tl_2,\ \mathsf{star\text{-}type}(Tp))\\ |\ \mathsf{fit\text{-}lists}(Tl_1,\ \mathsf{star\text{-}type}(N_2) - <\ >|\ Tl_2\ ,\ \mathsf{star\text{-}type}(Tp))\end{array}} \quad \text{[UFl10]}$$

$$\mathsf{fit\text{-}lists}(Tl_1,\ Tl_2,\ Tp)\ =\ \mathsf{false}\quad \textbf{otherwise} \quad \text{[UFl]}$$

5.2.5 Auxiliary modules

In Appendix B the auxiliary module **Orth-Booleans** is listed. Here, the composition of Boolean values and error messages is defined, as well as the necessary Boolean functions.

5.3 Extensions

5.3.1 Crucial features of ASF+SDF that might change

We are using features of ASF+SDF, of which some may be considered shortcomings, so ASF+SDF might be changed in a way that seriously affects this test.

Injections and subsorts in AsFix

The current format of AsFix terms contains all subsort information explicitly, in the sense that for every injection function, an extra Node is included. Until now, we have no serious reason to believe this will change, but it is useful to be aware of these 'extra' Nodes playing a crucial role in the fitting process.

List constructions as output sorts

There has been talk about admitting list constructions as the output sort of context-free functions. Most ASF+SDF users will be thrilled by this feature that eliminates the need for many conditions. Since the output sort of metavariables can be a list construction, but we have to treat them as functions, we already implemented this possibility as generally as possible in our unifiable test.
However, the comparing of two function Nodes uses syntactic equality (Equation UFo3). The possibility of fitting two function Nodes of the same list type, where one is of the '$*$' list type and the other of the '$+$' list type, is not yet included.

5.3.2 Specifications with default equations

Little change is needed to extend this test to ASF+SDF specifications with default equations. We can view the default property as a special kind of negative condition and forget all about it for the left-linearity test. And since default equations are only used for reduction in ASF+SDF, they can only overlap with each other. However, since the theoretical support for this type of conditional rewrite rules in CTRS theory is totally absent, we do not dare to say anything definite.

Chapter 6

Checking the property *recursive program scheme*

6.1 The property *recursive program scheme*

The property *recursive program scheme* tells us that all rules in a TRS satisfy a certain format. It implies the property orthogonal, which implies the property Church-Rosser (see Chapter 8). In such a TRS a term can always be reduced to the same term, regardless the application order of the rules.

Definition 6.1 *The TRS derived from an unconditional* ASF+SDF *specification that does not contain default equations is a* recursive program scheme *if:*

- *The syntax we use consists of a finite set of function symbols $\mathcal{F} = \{F_1, ..., F_n\}$ (the 'unknown' functions), where F_i either has arity $m_i \geq 0$ $(i = 1, ..., n)$ or is a list construction, and a finite set of function symbols $\mathcal{G} = \{G_1, ..., G_k\}$ (the 'known' or 'basic' functions), which is disjoint from $\mathcal{F}$ and where G_j either has arity $p_j \geq 0$ $(j = 1, ..., k)$ or is a list construction.*

- *Every rule is of the form:*
  $F_i(x_1, ..., x_{m_i}) \longrightarrow t_i$
  where all variables $x_1, ..., x_{m_i}$ are pairwise different and where t_i is an arbitrary term built from operators in $\mathcal{F}$, $\mathcal{G}$ and the variables $x_1, ..., x_{m_i}$.

- *For each F_i $(i = 1, ..., n)$ there is exactly one rule.*

Example. The following TRS is a recursive program scheme:

$$
\begin{array}{lcll}
\text{"F1"} & : & first(x, y, z) & \longrightarrow & x \\
\text{"M1"} & : & middle(x, y, z) & \longrightarrow & y \\
\text{"L1"} & : & last(x, y, z) & \longrightarrow & z
\end{array}
$$

The definition conforms to [Klo92], except for a slight modification concerning list constructions. Since the property is only defined for unconditional TRSs, we report failure if we encounter a condition or a label that indicates the use of a default equation in the ASF+SDF specifications.

6.2 Implementation

The implementation is very straightforward. The set of 'unknown' functions is built while inspecting the rules.

6.2.1 Module RPS

The main function rps only performs the test on the translation of the AsFix input if the original specification is unconditional.

$$\mathsf{rps}(T) \;=\; \mathsf{rps}(\mathsf{asfix2tree}(T)) \qquad \text{[RRp1]}$$

$$\mathsf{rps}(R^*) \;=\; \mathsf{if}(\mathsf{check\text{-}conditions}(R^*),\, \mathsf{rps\text{-}check}(R^*,\,)) \qquad \text{[RRp2]}$$

6.2.2 Module RPS-Check

Function rps-check checks all rules in the TRS one by one. First we let check-head ensure that the head symbol of the first rule's left-hand side is a function symbol. This means it can be either a lexeme, a metavariable, a function symbol with fixed arity or a list construction.
Then we let check-vars find out whether all children are variables, and pairwise different.
If these checks all succeed, we know this head symbol has to be in the set of 'unknown' functions $\mathcal{F}$, so we add it to the second argument and proceed with the rest of the rules.
The second argument is a Tree-List containing 'unknown' function symbols found in earlier rules.
When no more rules are left, we let check-func check the list of 'unknown' function symbols to satisfy the third condition of Definition 6.1.

$$\begin{array}{l} \mathsf{rps\text{-}check}(,\, Tl) = \qquad \text{[RRc1]} \\ \;\mathsf{if}(\mathsf{check\text{-}func}(Tl), \\ \quad \mathsf{true} \leftrightarrow \mathsf{The\ unknown\ functions\ are{:}}\ Tl, \\ \quad \mathsf{false} \leftrightarrow \mathsf{More\ than\ one\ equation\ for\ unknown\ functions}) \end{array}$$

$$\begin{array}{l} \mathsf{rps\text{-}check}(L : N - < Tl_1 > \longrightarrow Tr\ \mathsf{if}\ C^*\ R^*,\, Tl_2) = \qquad \text{[RRc2]} \\ \;\mathsf{if}(\mathsf{check\text{-}head}(N),\, \mathsf{if}(\mathsf{check\text{-}vars}(Tl_1),\, \mathsf{rps\text{-}check}(R^*,\, Tl_2 \mid N - < \ >), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{false} \leftrightarrow \mathsf{Wrong\ argument\ list\ in\ lhs\ eq{:}}\ L), \\ \qquad\qquad\qquad\quad \mathsf{false} \leftrightarrow \mathsf{No\ functional\ lhs\ in\ eq{:}}\ L) \end{array}$$

$$\mathsf{check\text{-}head}(N) \;=\; \mathsf{not}(\mathsf{var}(N)) \qquad \text{[RCh1]}$$

$$\begin{array}{lcll} \mathsf{check\text{-}func}() & = & \mathsf{true} & \text{[RCf1]} \\ \mathsf{check\text{-}func}(N - < \ > \mid Tl) & = & \mathsf{not}(\mathsf{in}(N,\, Tl))\ \&\ \mathsf{check\text{-}func}(Tl) & \text{[RCf2]} \end{array}$$

$$\begin{array}{lclll} \mathsf{check\text{-}vars}() & = & \mathsf{true} & & \text{[RCv1]} \\ \mathsf{check\text{-}vars}(N - < \ > \mid Tl) & = & \mathsf{var}(N)\ \&\ \mathsf{not}(\mathsf{in}(N,\, Tl))\ \&\ \mathsf{check\text{-}vars}(Tl) & & \text{[RCv2]} \\ \mathsf{check\text{-}vars}(Tl) & = & \mathsf{false} & \textbf{otherwise} & \text{[RCv]} \end{array}$$

6.2.3 Auxiliary modules

In Appendix C the auxiliary module **RPS-Booleans** is listed. Here, the composition of Boolean values and error messages is defined, as well as the necessary Boolean functions.

6.3 Extensions

6.3.1 List constructions as variable argument

An ASF+SDF term f(*Int**) may look like the typical left-hand side that satisfies the third condition of Definition 6.1, but it represents the term: f(“”(*xl*)). Since f's argument is now a list construction and not a variable, the test will fail.
If we want to be able to treat this kind of list construction as a variable argument, the following would have to be added to the equations of check-vars, making a few auxiliary functions necessary:

$$\mathsf{check\text{-}vars}(N - < Tl^{+} >|\ Tl) = \qquad [\mathsf{RCv3}]$$
$$\mathsf{list}(N)\ \&\ \mathsf{one\text{-}var}(Tl^{+})\ \&\ \mathsf{not}(\mathsf{in}(N - < Tl^{+} >\ ,\ Tl))\ \&\ \mathsf{check\text{-}vars}(Tl)$$

However, this might not be what we want, since in every test we have considered the list construction to be a function application, regardless the arguments, which conforms to the AsFix representation. So this will remain a point of discussion for now.

6.3.2 Conditional specifications

An extension of Definition 6.1 for conditional TRSs can be constructed easily when we ignore the conditions like we did in the orthogonality test. However, having conditions when only one rule per ‘unknown’ function symbol is allowed, might not be very sensible since it restricts the rewrite power and leaves you with (possibly) unwanted normal forms.
On the other hand, we might permit more rules per ‘unknown’ function symbol, under the restriction that the conditions are completely disjoint. That is: in each case only one of the rules with the same ‘unknown’ function symbol can be applied. This would involve semantic testing of conditions, which is not desirable and probably very hard.

Chapter 7

Checking the property *strongly normalizing*

7.1 The property *strongly normalizing*

The property *strongly normalizing* ensures that for every term, every reduction sequence eventually ends in a normal form. This very useful knowledge, since there is no way to stop the reduction process in the ASF+SDF Meta-environment.

Definition 7.1 *The TRS derived from an* ASF+SDF *specification is* strongly normalizing *(SN) if there is no infinite sequence* $t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow \ldots$. $\quad (t_i \in \mathcal{T}(\Sigma))$

Example. The following TRS is not SN:

$$\text{“Foo1”}: \quad \mathsf{foo}(1, x, y) \longrightarrow \mathsf{foo}(x, x, x)$$

because the term $\mathsf{foo}(1,1,2)$ causes an infinite rewrite sequence

$$\mathsf{foo}(1,1,2) \longrightarrow \mathsf{foo}(1,1,1) \longrightarrow \mathsf{foo}(1,1,1) \longrightarrow \ldots$$

In general, the property SN is undecidable([Klo92] Section 2.6). However, several procedures exist that can verify for a number of TRSs that they are SN. Since these procedures are not able to prove that a TRS is *not* SN, the answer of such a procedure will always be either ‘Yes’ or ‘I don’t know’.
We implemented such a procedure using the following intuition: If we can prove for each rule $l \longrightarrow r$ in a TRS R that r is smaller than l in some order, then we know that the reduction relation $\longrightarrow$ is descending with every step. If we can prove that the order is also well founded, then we can never have an infinite sequence, and the R is SN.
The strategy is called the *Recursive Path Ordering (RPO).*

We will first explain our definition of the RPO. Then we will show the relation between this definition and Klop’s RPO [Klo92], thus proving that when our test finds a TRS to be SN, then it could also be proved using Klop’s notion.
Having a sound theoretical basis, we will explain our implementation of the RPO in detail, and finally we dwell on covering of our test, that is, what type of rewrite rules cannot be handled, and in what way is it possible to extend the test.

7.1.1 The Recursive Path Ordering

The difference between the following definition of the Recursive Path Ordering, and the one in [Klo92] is that we define the reduction relation on the terms themselves, instead of defining it on terms mapped with natural numbers. The reason for this is that we started out with the Recursive Path Ordering of [BKM89] which is defined on the terms themselves. We discovered the more powerful Recursive Path Ordering of [Klo92], when a big part of the job had already been implemented, and decided not to change that aspect of our implementation.

First of all, we knew that the implementation would have to be easy to debug. We thought it easier to trace errors when processing terms with normal names, than terms with just natural numbers.

Secondly, the approach of the implementation was to make one term equal to another by descending both terms and use a lot of recursion. Mapping to natural numbers while descending might not work because of the least element 0 and the impossibility to have a number in between e.g. 2 and 3. So we needed to find an order on the function symbols anyway, and we thought mapping them to natural numbers might always be done afterwards. The definition for such a mapping can be found in Definition 7.7.

The correctness of this approach is proved in Proposition 7.8, so here we are.

Definition 7.2 *Let $\mathcal{T}(\Sigma)^*$ be the set of terms from $\mathcal{T}(\Sigma)$, where some of the non-variable nodes may be marked with a single * and commutativity is allowed (the arguments may be permuted). So $\mathcal{T}(\Sigma) \subseteq \mathcal{T}(\Sigma)^*$*

Definition 7.3 *Let O be a well-founded partial order on Σ.
On $\mathcal{T}(\Sigma)^*$ we define a reduction relation $\rightsquigarrow$ as follows.*

1. *place marker at the top:*

$$F(t_1, \dots, t_n) \rightsquigarrow F^*(t_1, \dots, t_n) \quad (n \geq 0)$$

2. *make copies below lesser top:*

$$F > G \in O \Longrightarrow F^*(t_1, \dots, t_n) \rightsquigarrow G(F^*(t_1, \dots, t_n), \dots, F^*(t_1, \dots, t_n))$$

   ($j \geq 0$ copies of $F^*(t_1, \dots, t_n), n \geq 0$)

3. *push marker down:*

$$F^*(G(s_1, \dots, s_m), t_1, \dots, t_n) \rightsquigarrow F(G^*(s_1, \dots, s_m), \dots, G^*(s_1, \dots, s_m), t_1, \dots, t_n)$$

   ($j \geq 0$ copies of $G^*(s_1, \dots, s_m), n, m \geq 0$)

4. *select argument:*

$$F^*(t_1, \dots, t_n) \rightsquigarrow t_i \quad (i \in \{1, \dots, n\}, n \geq 1)$$

5. *in context:*

$$s \rightsquigarrow t \Longrightarrow C[s] \rightsquigarrow C[t]$$

($s, s_1, \dots, s_m, t, t_1, \dots, t_n \in \mathcal{T}(\Sigma)^*$ and F, G non-variable elements of Σ)

We write $\rightsquigarrow^+$ for the transitive (but not reflexive) closure of $\rightsquigarrow$.

Theorem 7.4 *The relation $\rightsquigarrow^+$, restricted to $\mathcal{T}(\Sigma)$, is a well-founded partial ordering. It is SN.*

Proof: Let $\mapsto^+$ denote Klop's Recursive Path Ordering. This $\mapsto^+$ is a well-founded partial ordering, so a relation that is a restriction of $\mapsto^+$ must also be a well-founded partial ordering. In proposition 7.8 it is shown that $\leadsto^+$ is such a restriction, by proving that every $\leadsto^+$-reduction implies the existence of a $\mapsto^+$-reduction on the same terms.
Any well-founded partial order $>$ is SN, since there can be no infinite sequence $s_1 > s_2 > s_3 > \cdots$.
□

So in order to prove that a TRS is SN, we need to prove that a reduction in this TRS corresponds to a $\leadsto^+$ reduction in$\mathcal{T}(\Sigma)$. In other words: we determine a well-founded partial order on finitely many function symbols in Σ and show that for every rule $l \longrightarrow r$: $l \leadsto^+ r$. Then, every $\longrightarrow$-reduction step can be imitated by one or more $\leadsto$-reduction steps, so there can never be an infinite $\longrightarrow^+$-reduction, because that would have to be imitated by an infinite $\leadsto^+$-reduction which is impossible because of $\leadsto^+$ being SN. In Section 7.2.4 the orders determined are proved to be well founded and defined on finitely many function symbols.

Example. To illustrate the notion, we will prove the following TRS to be SN:

$$\text{“Not1”} \;:\; \mathsf{not}(\mathsf{and}(x, y)) \longrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))$$

A suitable order for this TRS is: $\{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}$.
Now we can show that the TRS is SN, by proving:

$$\mathsf{not}(\mathsf{and}(x, y)) \leadsto^+ \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))$$

Proof:

$$\begin{array}{rcl}
\mathsf{not}(\mathsf{and}(x, y)) & \overset{1}{\leadsto} & \mathsf{not}^*(\mathsf{and}(x, y)) \\
& \overset{4}{\leadsto} & \mathsf{and}(x, y) \\
& \overset{1}{\leadsto} & \mathsf{and}^*(x, y) \\
& \overset{2}{\leadsto} & \mathsf{or}(\mathsf{and}^*(x, y), \mathsf{and}^*(x, y)) \\
& \overset{2,5}{\leadsto} & \mathsf{or}(\mathsf{not}(\mathsf{and}^*(x, y)), \mathsf{and}^*(x, y)) \\
& \overset{2,5}{\leadsto} & \mathsf{or}(\mathsf{not}(\mathsf{and}^*(x, y)), \mathsf{not}(\mathsf{and}^*(x, y))) \\
& \overset{4,5}{\leadsto} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(\mathsf{and}^*(x, y))) \\
& \overset{4,5}{\leadsto} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))
\end{array}$$

In Section 7.2.3 we will explain how this particular order is determined. Section 7.2.5 shows the actual ASF+SDF equations that do the job.

The following notion stems from [BKM89] and eases further proofs considerably.

Definition 7.5 *Let $s, t \in \mathcal{T}(\Sigma)$. We say that t is* embeddable *in s when $s \searrow^* t$, with $\searrow^*$ the reflexive, transitive closure of reduction-relation $\searrow$ on $\mathcal{T}(\Sigma)$.*
$\searrow$ is defined by the following rules:

1. $F(t_1, \ldots, t_n) \searrow t_i$ $\quad (n \geq 1,\ 1 \leq i \leq n)$
2. $s \searrow t \Longrightarrow C[s] \searrow C[t]$

We write $s \searrow^+ t$ for the transitive (but not reflexive) closure of $\searrow$.

Example. $\mathsf{f}(x)$ is embeddable in $\mathsf{g}(\mathsf{f}(\mathsf{g}(x)))$, since: $\mathsf{g}(\mathsf{f}(\mathsf{g}(x))) \overset{1}{\searrow} \mathsf{f}(\mathsf{g}(x)) \overset{1,2}{\searrow} \mathsf{f}(x)$

Proposition 7.6 *Let $s, t \in \mathcal{T}(\Sigma)$ such that t is a true subterm of s.*

$$s \text{ is embeddable in } t \Longrightarrow s \rightsquigarrow^+ t$$

Proof: Since $s \not\equiv t$ we need at least one $\searrow$-reduction step: $s \searrow^+ t$. Now we can use induction on the number of steps in this $\searrow^+$-reduction. We write $s \searrow^n t$ for a reduction $s \searrow^+ t$ that consists of n $\searrow$-reduction steps.
Induction Hypothesis: $s \searrow^n t \Longrightarrow s \rightsquigarrow^+ t \quad (n \leq n_0)$
Basic step: $s \searrow t$. We distinguish two cases:

- $s \equiv F(t_1, \ldots, t_n) \overset{1}{\searrow} t_i \equiv t$.
  Then $s \equiv F(t_1, \ldots, t_n) \overset{1}{\rightsquigarrow} F^*(t_1, \ldots, t_n) \overset{4}{\rightsquigarrow} t_i \equiv t$.

- $s \equiv C[F(t_1, \ldots, t_n)] \overset{1,2}{\searrow} C[t_i] \equiv t$.
  Then $s \equiv C[F(t_1, \ldots, t_n)] \overset{1,5}{\rightsquigarrow} C[F^*(t_1, \ldots, t_n)] \overset{4,5}{\rightsquigarrow} C[t_i] \equiv t$.

Induction step: Suppose $s \searrow^{n_0+1} t$. Then there exists some t_i such that $s \searrow^{n_0} t_i \searrow t$. From the Induction Hypothesis, we know that $s \searrow^{n_0} t_i$ implies $s \rightsquigarrow^+ t_i$, and $t_i \searrow t$ implies $t_i \rightsquigarrow^+ t$. So we have $s \rightsquigarrow^+ t_i \rightsquigarrow^+ t$, which equals $s \rightsquigarrow^+ t$. □

The two useful results of Proposition 7.6 are that in this case no order is needed on the symbols $\in \Sigma$, and that at least one $\rightsquigarrow$-reduction step is performed. In Section 7.2.4 we only need the first result, in Section 7.2.5 both are used.

7.1.2 Our approach related to Klop's RPO

For ease of debugging and readability, we implemented the $\rightsquigarrow$ relation on the terms as they are in the TRS, instead of labeling them with natural numbers. Therefore we will prove that our reduction is just a restricted version from Klop's $\mapsto^+$, which is defined on labeled trees.

In the following, $\mathcal{T}_{\mathbb{N}}$ denotes Klop's set of commutative finite trees with nodes labeled by natural numbers, and $\mathcal{T}^*_{\mathbb{N}}$ the set of such trees where some of the nodes may be marked with a *.

Definition 7.7 *Let O be a well-founded partial order on the non-variable elements of Σ. We define a mapping $\mathsf{lab}_O : \Sigma \rightarrow \mathbb{N}$ that maps each unmarked non-variable symbol on a natural number and each variable on a new variable $x \in \mathbb{N}$. It satisfies the following condition:*

$$F > G \in O \Longrightarrow \mathsf{lab}_O(F) > \mathsf{lab}_O(G)$$

Now we can expand lab_O over terms $\in \mathcal{T}(\Sigma)^*$ to get trees $\in \mathcal{T}^*_{\mathbb{N}}$ as follows:

i. $\mathsf{lab}_O(F^*) = \mathsf{lab}_O(F)^*$

ii. $\mathsf{lab}_O(F(t_1, \ldots, t_n)) = \mathsf{lab}_O(F)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n))$

iii. $\mathsf{lab}_O(F^*(t_1, \ldots, t_n)) = \mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n))$

$(t_i, \ldots, t_n \in \mathcal{T}(\Sigma)^*$ *and F, G non-variable elements of* $\Sigma)$

A well-founded partial order can be extended to a total well-founded order, so we can extend O to the total well-founded order O'. If O is defined on finitely many function symbols, so is O'. Now we can easily construct $\mathsf{lab}_{O'}$, by 'topological sorting' (see [Knu73]).
$\mathsf{lab}_O = \mathsf{lab}_{O'}$ satisfies the condition posed above.

Proposition 7.8 *Let O be a well-founded partial order on finitely many non-variable symbols in Σ and lab_O a mapping as in Definition 7.7. Then*

$$s \rightsquigarrow^+ t \Longrightarrow \mathsf{lab}_O(s) \mapsto^+ \mathsf{lab}_O(t)$$

Proof: By induction on the number of steps in $\rightsquigarrow^+$-reductions.
We write $s \rightsquigarrow^n t$ for a reduction $s \rightsquigarrow^+ t$ that consists of n $\rightsquigarrow$-reduction steps, and $u \mapsto^n v$ for a reduction $u \mapsto^+ v$ that consists of n $\mapsto$-reduction steps.
$(s, t \in \Sigma,\ u, v \in \mathcal{T}_{\mathbb{N}})$
Induction Hypothesis: $s \rightsquigarrow^n t \Longrightarrow \mathsf{lab}_O(s) \mapsto^n \mathsf{lab}_O(t) \quad (n \leq n_0)$
Basic step: $s \rightsquigarrow t \Longrightarrow \mathsf{lab}_O(s) \mapsto \mathsf{lab}_O(t)$
We prove this for every possible $\rightsquigarrow$-reduction step:

1. $s \equiv F(t_1, \ldots, t_n) \rightsquigarrow F^*(t_1, \ldots, t_n) \equiv t$. Now

$$\begin{aligned}
\mathsf{lab}_O(s) &\equiv \mathsf{lab}_O(F(t_1, \ldots, t_n)) \\
&\overset{7.7(iii)}{=} \mathsf{lab}_O(F)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n)) \\
&\overset{i}{\mapsto} \mathsf{lab}_O(F)^*(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n)) \\
&\overset{7.7(ii)}{=} \mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n)) \\
&\overset{7.7(iv)}{=} \mathsf{lab}_O(F^*(t_1, \ldots, t_n)) \\
&\equiv \mathsf{lab}_O(t)
\end{aligned}$$

2. $F > G \in O \Longrightarrow s \equiv F^*(t_1, \ldots, t_n) \rightsquigarrow G(F^*(t_1, \ldots, t_n), \ldots, F^*(t_1, \ldots, t_n)) \equiv t$. Because of 7.7(i) we know that $\mathsf{lab}_O(F) > \mathsf{lab}_O(G)$, so

$$\begin{aligned}
\mathsf{lab}_O(s) &\equiv \mathsf{lab}_O(F^*(t_1, \ldots, t_n)) \\
&\overset{7.7(iv)}{=} \mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n)) \\
&\overset{7.7(ii)}{=} \mathsf{lab}_O(F)^*(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n)) \\
&\overset{ii}{\mapsto} \mathsf{lab}_O(G)(\mathsf{lab}_O(F)^*(\mathsf{lab}_O(t_1), \ldots,\ \mathsf{lab}_O(t_n)), \ldots, \\
&\qquad \mathsf{lab}_O(F)^*(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n))) \\
&\overset{7.7(ii)}{=} \mathsf{lab}_O(G)(\mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1), \ldots,\ \mathsf{lab}_O(t_n)), \ldots, \\
&\qquad \mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1), \ldots, \mathsf{lab}_O(t_n))) \\
&\overset{7.7(iv)}{=} \mathsf{lab}_O(G)(\mathsf{lab}_O(F^*(t_1, \ldots, t_n)), \ldots,\ \mathsf{lab}_O(F^*(t_1, \ldots, t_n))) \\
&\overset{7.7(vi)}{=} \mathsf{lab}_O(G(F^*(t_1, \ldots, t_n)), \ldots, (F^*(t_1, \ldots, t_n))) \\
&\equiv \mathsf{lab}_O(t)
\end{aligned}$$

3. $s \equiv F^*(G(s_1, \ldots, s_m), t_1, \ldots, t_n) \rightsquigarrow F(G^*(s_1, \ldots, s_m), \ldots, G^*(s_1, \ldots, s_m), t_1, \ldots, t_n) \equiv t$.

$$\mathsf{lab}_O(s) \quad \equiv \quad \mathsf{lab}_O(F^*(G(s_1, \ldots, s_m), t_1, \ldots, t_n)$$

$$
\begin{aligned}
&\overset{7.7(iv)}{=} && \mathsf{lab}_O(F^*)(\mathsf{lab}_O(G(s_1,\ldots,s_m)),\ \mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(iv)}{=} && \mathsf{lab}_O(F^*)(\mathsf{lab}_O(G)(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(ii)}{=} && \mathsf{lab}_O(F)^*(\mathsf{lab}_O(G)(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{iii}{\mapsto} && \mathsf{lab}_O(F)(\mathsf{lab}_O(G)^*(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\ldots,\mathsf{lab}_O(G)^*(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\\
& && \qquad \mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(ii)}{=} && \mathsf{lab}_O(F)(\mathsf{lab}_O(G^*)(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\ldots,\mathsf{lab}_O(G^*)(\mathsf{lab}_O(s_1),\ldots,\ \mathsf{lab}_O(s_m)),\\
& && \qquad \mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(iv)}{=} && \mathsf{lab}_O(F)(\mathsf{lab}_O(G^*(s_1,\ldots,s_m)),\ldots,\ \mathsf{lab}_O(G^*(s_1,\ldots,s_m)),\mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(iv)}{=} && \mathsf{lab}_O(F(G^*(s_1,\ldots,s_m),\ldots,G^*(s_1,\ldots,s_m),t_1,\ldots,t_n))\\
&\equiv && \mathsf{lab}_O(t)
\end{aligned}
$$

It is obvious that while doing $\overset{iii}{\mapsto}$, we take exactly the number of copies of $\mathsf{lab}_O(G)(\mathsf{lab}_O(s_1),\ldots,\mathsf{lab}_O(s_m))$ that is needed in t, say j copies. Then the following $\overset{7.7(ii)}{=}$ and $\overset{7.7(iv)}{=}$ are naturally performed j times (one for each copy).

4. $s \equiv F^*(t_1,\ldots,t_n) \rightsquigarrow t_i \equiv t$. This is easy:

$$
\begin{aligned}
\mathsf{lab}_O(s) &\equiv \mathsf{lab}_O(F^*(t_1,\ldots,t_n))\\
&\overset{7.7(iv)}{=} \mathsf{lab}_O(F^*)(\mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{7.7(ii)}{=} \mathsf{lab}_O(F)^*(\mathsf{lab}_O(t_1),\ldots,\mathsf{lab}_O(t_n))\\
&\overset{iv}{\mapsto} \mathsf{lab}_O(t_i)\\
&\equiv \mathsf{lab}_O(t)
\end{aligned}
$$

5. $s \equiv C[s'] \rightsquigarrow C[t'] \equiv t$, because $s' \rightsquigarrow t'$. Suppose s' equals one of the cases above. Then we can conclude that $\mathsf{lab}_O(s') \mapsto \mathsf{lab}_O(t')$, and prove with induction on the depth of the context $C[]$ that $\mathsf{lab}_O(C[s']) \overset{v}{\mapsto} \mathsf{lab}_O(C[t'])$. This is left to the reader.
If s' does not equal one of the cases above, we can find a context $C'[]$ such that $s \equiv C'[u] \rightsquigarrow C'[v] \equiv t$ and u equals one of the cases above. $(u, v \in \mathcal{T}(\Sigma))$

$(s, s_1,\ldots,s_m,t,t_i,\ldots,t_n \in \mathcal{T}(\Sigma)^*$ and F, G non-variable elements of $\Sigma)$

Induction step:
Suppose $s \rightsquigarrow^{n_0+1} t$. Then there exists some t_i such that $s \rightsquigarrow^{n_0} t_i \rightsquigarrow t$. From the Induction Hypothesis, we know that $s \rightsquigarrow^{n_0} t_i$ implies $\mathsf{lab}_O(s) \mapsto^{n_0} \mathsf{lab}_O(t_i)$, and $t_i \rightsquigarrow t$ implies $\mathsf{lab}_O(t_i) \mapsto \mathsf{lab}_O(t)$. So we have $\mathsf{lab}_O(s) \mapsto^{n_0} \mathsf{lab}_O(t_i) \mapsto \mathsf{lab}_O(t)$, which equals $\mathsf{lab}_O(s) \mapsto^{n_0+1} \mathsf{lab}_O(t)$. □

7.2 Implementation

General approach

The strategy we use consists of two steps: first we find all satisfactory orders, as explained below. Then, for every rule $l \longrightarrow r$ we try to $\rightsquigarrow^+$-rewrite l to r.

The trees we use in the implementation are not explicitly commutative. In the establishing of the orders, and during the reduction process, the argument list of a function node is permuted when necessary, thus simulating the commutativity in a satisfactory manner. This is done in module **Find-Relation** (Section 7.2.4) and in module **RPO-Rewrite** (Section 7.2.5).
Since metavariables are considered to be syntactical constants (see Section 3.3), we treat them just like the other markable symbols.

Establishing possible orders

We use {} to indicate the empty partial order that is our starting point. If it is not extended, we obviously can $\rightsquigarrow^+$-reduce using just rules *1*,*3*,*4* and *5*. If, for some reason, we cannot establish any well-founded order, we indicate this by having found no order at all (the empty order sequence). Note the difference between {} and the empty order sequence!

When trying to establish the partial orders that might lead to successful $\rightsquigarrow^+$-reduction, we have some useful knowledge:
We know what the top parts of the current tree and the goal tree look like, and we know what sort of rules are used during the reduction. This knowledge leads to certain choices for the orders. The explanation of these choices and an example can be found in Section 7.2.4.

In an order sequence, orders are separated by #.

RPO-reduction

An example can be found in Section 7.2.5.

7.2.1 Module SN

Function sn will translate ASF+SDF input to a TRS. Then, if the rules do not contain conditions or 'default' labels, get-orders and rpo-check are allowed to find the partial orders and perform RPO-reduction.

$$\mathsf{sn}(T) \ = \ \mathsf{sn}(\mathsf{asfix2tree}(T)) \qquad [\mathsf{SSn1}]$$

$$\mathsf{sn}(R^*) = \mathsf{if}(\mathsf{check\text{-}conditions}(R^*),\ \mathsf{rpo\text{-}check}(R^*,\ \mathsf{get\text{-}orders}(R^*,\ \{\}))) \qquad [\mathsf{SSn2}]$$

7.2.2 Module RPO-Check

Function rpo-check tries to find an order in the sequence of orders, that permits successful $\rightsquigarrow^+$-reduction on all rules. The lazy implementation of the Boolean '|' ensures that we stop as soon as a proper order is found.

$$\mathsf{rpo\text{-}check}(R^*,\ ) \ = \ \mathsf{false} \leftrightarrow \mathsf{All\ orders\ failed} \qquad [\mathsf{RCh1}]$$

$$\mathsf{rpo\text{-}check}(R^*,\ O\#\ O^*) \ = \ \mathsf{check\text{-}per\text{-}order}(R^*,\ O) \mid \mathsf{rpo\text{-}check}(R^*,\ O^*) \qquad [\mathsf{RCh2}]$$

Function check-per-order takes every rule and lets reduce do the $\rightsquigarrow^+$-reduction on the left-hand side, using the right-hand side as stop condition.

$$\text{check-per-order}(,\ O)\ =\ \text{true} \leftrightarrow \text{All equations checked using order} : O \qquad \text{[RCp1]}$$

$$\begin{aligned}&\text{check-per-order}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^*\ R^*,\ O) = \qquad \text{[RCp2]}\\ &\ \text{if}(\text{reduce}(\text{tree2mtree}(Tr_1),\ \text{tree2mtree}(Tr_2),\ O),\\ &\quad \text{check-per-order}(R^*,\ O),\\ &\quad \text{false} \leftrightarrow \text{Failed reducing eq} : L)\end{aligned}$$

7.2.3 Module Get-Orders

We want get-orders to find the most general partial orders that might lead to success. As long as there are rules left, and the order sequence of already established orders is not empty, we look at the first rule in the set of rules. If its right-hand side is embeddable in its left-hand side, we will not need already found relations or new relations to $\rightsquigarrow^+$-reduce this rule (Proposition 7.6) and we can proceed with the remaining set of rules and the unchanged order sequence. Otherwise, we will let find-relations add new relations to the current order sequence. Since find-relations may forbid some of the orders that were found earlier, and keep others, we use just the result order sequence for the remaining set of rules.
As mentioned before, the empty order sequence is an indication of something wrong that cannot be solved. It is a stop condition.

$$\text{get-orders}(,\ O^+)\ =\ O^+ \qquad \text{[GGo1]}$$

$$\text{get-orders}(R^*,\ )\ = \qquad \text{[GGo2]}$$

$$\frac{\text{embeddable}(Tr_2,\ Tr_1) = \text{true}}{\text{get-orders}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^*\ R^*,\ O^+) = \text{get-orders}(R^*,\ O^+)} \qquad \text{[GGo3]}$$

$$\frac{\begin{array}{c}\text{embeddable}(Tr_2,\ Tr_1) = \text{false},\\ \text{find-relations}(Tr_1,\ Tr_2,\ O^+) = O^*\end{array}}{\text{get-orders}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^*\ R^*,\ O^+) = \text{get-orders}(R^*,\ O^*)} \qquad \text{[GGo4]}$$

7.2.4 Module Find-Relation

Function find-relations implements a naive $\rightsquigarrow^+$-reduction. Doing just the relational work, we forget about the * marking business for the time being. We know that we will only get more possibilities and save ourselves a lot of trouble.

Find-relations is concerned with discovering all possible reduction paths that reduce the current Tree (first argument) to the goal Tree (second argument) and returning the most general partial orders needed for every successful path.
We try to reach the goal Tree by descending both Trees and leave identical Nodes above.

Example.

For readability, this example uses the CTRS notation instead of the Tree format.
Using TRS: "Not1" : $\mathsf{not}(\mathsf{and}(x,y)) \longrightarrow \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$
the term

$$\text{find-relations(} \mathsf{not}(\mathsf{and}(x,y)),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

reduces to $\{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\#\{\mathsf{not} > \mathsf{or}\}$.
We start with

$$\text{find-relations(} \mathsf{not}(\mathsf{and}(x,y)),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

Since the top Nodes are not equal and the order is empty we will have to apply Equation FFr4. We will assume for the scope of this example that in such an equation the variables O^* can be substituted by the other side of the corresponding condition, supposing that the conditions succeed. Then we can reduce to

$$\text{find-left(} \mathsf{and}(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

$$\#\ \text{every-right(} \mathsf{not}(\mathsf{and}(x,y)),\ \mathsf{not}(x),\mathsf{not}(y)\ ,\ \{\mathsf{not} > \mathsf{or}\}\ )$$

$$\#\ \text{find-relations(} \mathsf{not}(\mathsf{and}(x,y)),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ )\qquad (1)$$

The third argument of # will reduce to the empty sequence, so we remove that part. To keep the view clear, we will now reduce the first argument of #. and then the second argument. So we take

$$\text{find-left(} \mathsf{and}(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

Using Equation FFl1, we reduce to

$$\text{find-relations(} \mathsf{and}(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

$$\#\ \text{find-left(, } \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

The second argument of # will reduce to the empty sequence, so we remove that argument. Again, the top Nodes are not equal and the order is still empty, so we apply Equation FFr4 and reduce to

$$\text{find-left(} x,y\ ,\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\ \}\ )$$

$$\#\ \text{every-right(} \mathsf{and}(x,y),\ \mathsf{not}(x),\mathsf{not}(y)\ ,\ \{\mathsf{and} > \mathsf{or}\}\ )$$

$$\#\ \text{find-relations(} \mathsf{and}(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ )\qquad (2)$$

The third argument of # will reduce to the empty sequence, so we remove that argument. The first argument will also reduce to the empty sequence, since we cannot form a Tree with non-variable Nodes out of a variable list. So we remove that argument too and keep only the second argument of #. That can be reduced by applying Equation FEr2, if we successfully reduce the following:

$$\text{find-relations(} \mathsf{and}(x,y),\ \mathsf{not}(x),\ \{\mathsf{and} > \mathsf{or}\}\ )$$

We apply Equation FFr4 and reduce to

$$\text{find-left(} x,y\ ,\ \mathsf{not}(x),\ \{\mathsf{and} > \mathsf{or}\}\ )$$

$$\#\ \text{every-right(} \mathsf{and}(x,y),\ x,\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

$$\#\ \text{find-relations(} \mathsf{and}(x,y),\ \mathsf{not}(x),\ )\qquad (3)$$

The first and third argument will reduce to the empty sequence, as seen before, so we remove those arguments. Again we can reduce the second argument by applying Equation FEr2, if we successfully reduce the following:

$$\textsf{find-relations}(\ \textsf{and}(x,y),\ x,\ \{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}\ )$$

Since x is embeddable in $\textsf{and}(x,y)$, we can apply Equation FFr1 and reduce to

$$\{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}$$

Now, *(3)* reduces to

$$\textsf{every-right}(\ \textsf{and}(x,y),\ ,\ \{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}\ )$$

Applying Equation FEr1 we reduce to

$$\{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}$$

Now, *(2)* reduces to

$$\textsf{every-right}(\ \textsf{and}(x,y),\ \textsf{not}(y),\ \{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}\ )$$

Performing the same steps again, we can reduce to

$$\textsf{every-right}(\ \textsf{and}(x,y),\ ,\ \{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}\ )$$

Applying Equation FEr1 we reduce to

$$\{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}$$

We still have to reduce one # argument of *(1)*, namely

$$\textsf{every-right}(\ \textsf{not}(\textsf{and}(x,y)),\ \textsf{not}(x), \textsf{not}(y)\ ,\ \{\textsf{not} > \textsf{or}\}\ ) \qquad (4)$$

This can be reduced by applying Equation FEr2, if we successfully reduce the following:

$$\textsf{find-relations}(\ \textsf{not}(\textsf{and}(x,y)),\ \textsf{not}(x),\ \{\textsf{not} > \textsf{or}\}\ )$$

Since $\textsf{not}(x)$ is embeddable in $\textsf{not}(\textsf{and}(x,y))$, we can apply Equation FFr1 and reduce to

$$\{\textsf{not} > \textsf{or}\}$$

Now *(4)* reduces to

$$\textsf{every-right}(\ \textsf{not}(\textsf{and}(x,y)),\ \textsf{not}(y),\ \{\textsf{not} > \textsf{or}\}\ ) \qquad (5)$$

This can be reduced by applying Equation FEr2, if we successfully reduce the following:

$$\textsf{find-relations}(\ \textsf{not}(\textsf{and}(x,y)),\ \textsf{not}(y),\ \{\textsf{not} > \textsf{or}\}\ )$$

Since $\textsf{not}(y)$ is embeddable in $\textsf{not}(\textsf{and}(x,y))$, we can apply Equation FFr1 and reduce to

$$\{\textsf{not} > \textsf{or}\}$$

Now *(5)* reduces to

$$\textsf{every-right}(\ \textsf{not}(\textsf{and}(x,y)),\ ,\ \{\textsf{not} > \textsf{or}\}\ )$$

Applying Equation FEr1, we reduce to

$$\{\textsf{not} > \textsf{or}\}$$

So the normal form of *(1)* is

$$\{\textsf{and} > \textsf{not}, \textsf{and} > \textsf{or}\}\ \#\ \{\textsf{not} > \textsf{or}\}$$

The ASF+SDF code

We have one stop condition indicating a successful path: the second Tree is embeddable in the first Tree. As shown in Proposition 7.6, no extension of the orders found is needed and the current order sequence describes all possibilities.

$$\frac{\mathsf{embeddable}(\mathit{Tr}_2,\ \mathit{Tr}_1) = \mathsf{true}}{\mathsf{find\text{-}relations}(\mathit{Tr}_1,\ \mathit{Tr}_2,\ O^+) = O^+} \qquad \text{[FFr1]}$$

When the 'embeddable' condition fails, we may try the three remaining equations.

The top Nodes are the same. We can try if some concatenation of rule *3* yields a successful path, by calling find-all-lefts. We can also apply rule *4* and let find-left find a proper child in the current Tree to reduce to the goal Tree. All of the found orders are put together in a new sequence.

$$\frac{\begin{array}{r}\mathsf{embeddable}(N - <\mathit{Tl}_2>,\ N - <\mathit{Tl}_1>) = \mathsf{false},\\ \mathit{Tl}_1 \neq \mathit{Tl}_2,\\ \mathsf{find\text{-}all\text{-}lefts}(\mathit{Tl}_1,\ \mathit{Tl}_2,\ O^+) = O_1^*,\\ \mathsf{find\text{-}left}(\mathit{Tl}_1,\ N - <\mathit{Tl}_2>,\ O^+) = O_2^*\end{array}}{\mathsf{find\text{-}relations}(N - <\mathit{Tl}_1>,\ N - <\mathit{Tl}_2>,\ O^+) = O_1^* \#\ O_2^*} \qquad \text{[FFr2]}$$

The top Nodes are not equal. We look at the first order in the sequence. In this order, the goal top Node is greater than the current top Node, making the application of rule *2* impossible. So all we can do obeying this order is apply rule *4* and let find-left find a proper child in the current Tree to reduce to the goal Tree. In the remaining orders, things might be different, so they are processed recursively.

$$\frac{\begin{array}{r}\mathsf{embeddable}(N_2 - <\mathit{Tl}_2>,\ N_1 - <\mathit{Tl}_1>) = \mathsf{false},\\ N_1 \neq N_2,\\ \mathsf{var}(N_1) = \mathsf{false},\\ \mathsf{var}(N_2) = \mathsf{false},\\ \mathsf{in}(N_2 > N_1,\ O) = \mathsf{true},\\ \mathsf{find\text{-}left}(\mathit{Tl}_1,\ N_2 - <\mathit{Tl}_2>,\ O) = O_1^*,\\ \mathsf{find\text{-}relations}(N_1 - <\mathit{Tl}_1>,\ N_2 - <\mathit{Tl}_2>,\ O^*) = O_2^*\end{array}}{\mathsf{find\text{-}relations}(N_1 - <\mathit{Tl}_1>,\ N_2 - <\mathit{Tl}_2>,\ O\#\ O^*) = O_1^* \#\ O_2^*} \qquad \text{[FFr3]}$$

The top Nodes are not equal, and the goal top Node is not greater than the current top Node in the first order in the sequence. We try everything we did in the previous case, but we may also add the relation expressing the current top Node to be smaller than the goal top Node to the first order. Now we can apply rule *2*, by keeping the current Tree, taking the children of the goal Tree and letting every-right make a copy from the current Tree for every Tree in the goal list.

$$\frac{\begin{array}{r}\mathsf{embeddable}(N_2 - <\mathit{Tl}_2>,\ N_1 - <\mathit{Tl}_1>) = \mathsf{false},\\ N_1 \neq N_2,\\ \mathsf{var}(N_1) = \mathsf{false},\\ \mathsf{var}(N_2) = \mathsf{false},\\ \mathsf{in}(N_2 > N_1,\ O) = \mathsf{false},\\ \mathsf{find\text{-}left}(\mathit{Tl}_1,\ N_2 - <\mathit{Tl}_2>,\ O) = O_1^*,\\ \mathsf{every\text{-}right}(N_1 - <\mathit{Tl}_1>,\ \mathit{Tl}_2,\ \mathsf{trans\text{-}close}(\mathsf{add}(N_1 > N_2,\ O))) = O_2^*,\\ \mathsf{find\text{-}relations}(N_1 - <\mathit{Tl}_1>,\ N_2 - <\mathit{Tl}_2>,\ O^*) = O_3^*\end{array}}{\mathsf{find\text{-}relations}(N_1 - <\mathit{Tl}_1>,\ N_2 - <\mathit{Tl}_2>,\ O\#\ O^*) = O_1^* \#\ O_2^* \#\ O_3^*} \qquad \text{[FFr4]}$$

When all else fails, we return the empty order sequence.

$$\text{find-relations}(\mathit{Tr}_1,\ \mathit{Tr}_2,\ O^*)\ =\quad \textbf{otherwise} \qquad \text{[FFr]}$$

Function find-left must find one or more Trees in the current list that can be successfully reduced to the goal Tree. So for every Tree in the current list, the sequence of possibly successful orders is computed, and all of these are put together in a new sequence.

$$\frac{\begin{array}{c}\text{find-relations}(\mathit{Tr}_1,\ \mathit{Tr}_2,\ O^+) = O_1^*,\\ \text{find-left}(\mathit{Tl},\ \mathit{Tr}_2,\ O^+) = O_2^*\end{array}}{\text{find-left}(\mathit{Tr}_1|\ \mathit{Tl},\ \mathit{Tr}_2,\ O^+) = O_1^* \#\ O_2^*} \qquad \text{[FFl1]}$$

$$\text{find-left}(\mathit{Tl},\ \mathit{Tr},\ O^*)\ =\quad \textbf{otherwise} \qquad \text{[FFl]}$$

Function every-right wants to find an order sequence for successfully reducing the current Tree to every Tree in the goal list. So if a non-empty order sequence is found for the first Tree in the goal list, we take this intermediate sequence along for the rest of the goal list.

$$\text{every-right}(\mathit{Tr}_1,\ ,\ O^+)\ =\ O^+ \qquad \text{[FEr1]}$$

$$\frac{\text{find-relations}(\mathit{Tr}_1,\ \mathit{Tr}_2,\ O^+) = O^{+'}}{\text{every-right}(\mathit{Tr}_1,\ \mathit{Tr}_2|\ \mathit{Tl},\ O^+) = \text{every-right}(\mathit{Tr}_1,\ \mathit{Tl},\ O^{+'})} \qquad \text{[FEr2]}$$

$$\text{every-right}(\mathit{Tr},\ \mathit{Tl},\ O^*)\ =\quad \textbf{otherwise} \qquad \text{[FEr]}$$

Function find-all-lefts is actually a combination of both find-left and every-right. It has to produce the goal list, given the current list. Since we want to express a possible concatenation of rule *3*, we do not worry at this point about which Trees can be multiplied and which Trees have to be copied exactly once. We just let find-left find proper candidates for every Tree in the goal list.

$$\text{find-all-lefts}(\mathit{Tl},\ ,\ O^+)\ =\ O^+ \qquad \text{[FMa1]}$$

$$\frac{\text{find-left}(\mathit{Tl}^+,\ \mathit{Tr},\ O^+) = O^{+'}}{\text{find-all-lefts}(\mathit{Tl}^+,\ \mathit{Tr}|\ \mathit{Tl},\ O^+) = \text{find-all-lefts}(\mathit{Tl}^+,\ \mathit{Tl},\ O^{+'})} \qquad \text{[FMa2]}$$

$$\text{find-all-lefts}(\mathit{Tl}_1,\ \mathit{Tl}_2,\ O^*)\ =\quad \textbf{otherwise} \qquad \text{[FMa]}$$

An important issue must still be dealt with. We said before we would prove our orders to be well founded and defined on finitely many symbols. At this point it is very easy to see this is indeed the case.

Proposition 7.9 *The orders established by function* find-relations *are all well-founded partial orders defined on finitely many function symbols.*

Proof: The orders are only changed in Equation FFr4, in the other equations they are just passed on or destroyed. Suppose that every find-relations-term reduces to normal form. This means that Equation FFr4 is applied a finite number of times and enables us to use induction on the orders that are built by Equation FFr4. We write O_{i+1} for the order, obtained by computing trans-close(add($F > G, O_i$)), where $i \geq 0$ and $F > G$ is a *legal* pair.
$O_0 = \{ \}$.
A pair $F > G$ is *legal* if $G > F \notin O_i$.
Induction Hypothesis: O_n is a well-founded partial order defined on finitely many symbols for $n = n_0$.
Basic step: { } is trivially well founded and defined on finitely many symbols (namely 0).
Induction step: O_{n_0+1} = trans-close(add($F > G, O_{n_0}$)). Since O_{n_0} is defined on finitely many symbols, the addition of pair $F > G$ causes the number of symbols to be increased with at most 2. A finite number of symbols $+k$, where k is 0, 1, or 2, is again a finite number of symbols, so O_{n_0+1} is defined on finitely many symbols. The addition of a pair $F > G$ to a well-founded order in which $G \not> F$ yields a well-founded order. Computing funtrans-close (the transitive closure) is only necessary for technical reasons, since mathematically speaking, an order is already closed under transitivity.
The termination of find-relations can be proved using the finiteness of the Tree arguments it gets, and the fact that recursive calls of find-relations carry smaller Tree arguments. This is left to the reader. □

7.2.5 Module RPO-Reduce

Function reduce tries to reduce the current Tree (first argument) to the goal Tree (second argument), using the set of rules for $\rightsquigarrow$. From this point we will use the Marked-Tree format, to be able to mark the Nodes of a Tree. Of course the goal Tree is never marked, but it is also translated to this new format for ease of comparing.
We try to reach the goal Tree by descending both Trees. So in most cases rule *5* will not be applied, since we do not descend unless we leave identical (unmarked!!) current and goal tops above.

Example.

For readability, this example uses the CTRS notation instead of the Tree format.
Considering TRS: "Not1" : $\mathsf{not}(\mathsf{and}(x, y)) \longrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))$
and the first order in the order sequence found by find-relations, the term

$$\mathsf{reduce}(\ \mathsf{not}(\mathsf{and}(x, y)),\ \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

reduces to true.
We will reproduce the reduction here, skipping a few unimportant reduction steps. Start with

$$\mathsf{reduce}(\ \mathsf{not}(\mathsf{and}(x, y)),\ \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

Nothing marked and no equal top Nodes, so using Equation RRe6 we reduce to

$$\mathsf{reduce}(\ \mathsf{not}^*(\mathsf{and}(x, y)),\ \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

Since not > or is not in our order, we can only try the second part of the Boolean '|' in Equation RRe7 that selects one of the children according to rule *4* and reduce to

$$\mathsf{false}\ |\ \mathsf{oneleft2right}(\ \mathsf{and}(x, y),\ \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

Remove the false | part. There is only one child we can choose, so using Equation ROl2 we reduce to

$$\mathsf{reduce}(\ \mathsf{and}(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )\ \mid\ \mathsf{false}$$

Remove the | false part. Nothing marked and no equal top Nodes, so using Equation RRe6 we reduce to

$$\mathsf{reduce}(\ \mathsf{and}^*(x,y),\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

Now and > or is in our order, so we can make two copies of $\mathsf{and}^*(x,y)$ under lesser symbol or. This means reduce $\mathsf{and}^*(x,y)$ to every child of or. Using Equation RRe7 we reduce to

$$\mathsf{left2everyright}(\ \mathsf{and}^*(x,y),\ \mathsf{not}(x),\mathsf{not}(y)\ ,\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

$$\mid\ \mathsf{oneleft2right}(\ x,y\ ,\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y)),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

The second argument of | will reduce to false since we cannot do anything with variables, so remove that part. Two applications of Equation RLe2 reduce the other argument to

$$\mathsf{reduce}(\ \mathsf{and}^*(x,y),\ \mathsf{not}(x),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

$$\&\ \mathsf{reduce}(\ \mathsf{and}^*(x,y),\ \mathsf{not}(y),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

Take the first argument of & and reduce it to true. The reader will hopefully be convinced that in case of success, the second argument will also reduce to true. So we have

$$\mathsf{reduce}(\ \mathsf{and}^*(x,y),\ \mathsf{not}(x),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

We have and > not in our order, so we can make one copy of $\mathsf{and}^*(x,y)$ under lesser symbol not. This means reduce $\mathsf{and}^*(x,y)$ to the one child of not. Using Equation RRe7 we reduce to

$$\mathsf{left2everyright}(\ \mathsf{and}^*(x,y),\ x,\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\})$$

$$\mid\ \mathsf{oneleft2right}(\ x,y\ ,\ \{\mathsf{not}(x),\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )$$

The second argument of | will reduce to false since we cannot do anything with variables, so remove that part. One application of Equation RLe2 reduces the other argument to

$$\mathsf{reduce}(\ \mathsf{and}^*(x,y),\ x,\ \{\mathsf{and} > \mathsf{not}, \mathsf{and} > \mathsf{or}\}\ )\ \&\ \mathsf{true}$$

Since we just need to reduce to a variable, we only need to know if it is a subterm of $\mathsf{and}^*(x,y)$. We use Equation RRe3 and reduce to

$$\mathsf{in}(x,\ \mathsf{and}^*(x,y))$$

Obviously, x is a subterm of $\mathsf{and}^*(x,y)$, so we reduce to

$$\mathsf{true}$$

The ASF+SDF code

A successful reduction is easily recognized: the current and goal Tree are equal. The goal Tree was never marked, so the current Tree has no marks either.

$$\mathsf{reduce}(\mathit{Trm},\ \mathit{Trm},\ O)\ =\ \mathsf{true} \qquad \text{[RRe1]}$$

The current Tree is just a variable, so it must be equal to the goal Tree, since no $\rightsquigarrow$-rule applies to variables.

$$\frac{\mathsf{var}(\mathit{Nm}_1) = \mathsf{true}}{\mathsf{reduce}(\mathit{Nm}_1 - < \ > , \mathit{Nm}_2 - < \mathit{Tlm} > , O) = \mathsf{eq}(\mathit{Nm}_1, \mathit{Nm}_2)} \quad \text{[RRe2]}$$

The goal Tree is just a variable, so it must be a subterm of the current Tree. If it is, we know that we can reach it by further reducing regardless the contents of order O, because of Proposition 7.6.

$$\frac{\mathsf{var}(\mathit{Nm}_2) = \mathsf{true}}{\mathsf{reduce}(\mathit{Nm}_1 - < \mathit{Tlm} > , \mathit{Nm}_2 - < \ > , O) = \mathsf{in}(\mathit{Nm}_2, \mathit{Nm}_1 - < \mathit{Tlm} >)} \quad \text{[RRe3]}$$

The top Nodes are equal and unmarked, so we can apply rule *1* and mark the current top Node, or try rule *5* and reduce in context by taking the current and goal children list. Since permutations of the arguments are allowed during $\rightsquigarrow$-rewriting, map-permutations tries every possible one.

$$\frac{\begin{array}{c}\mathsf{var}(\mathit{Nm}) = \mathsf{false}, \\ \mathit{Tlm}_1 \neq \mathit{Tlm}_2\end{array}}{\begin{array}{l}\mathsf{reduce}(\mathit{Nm} - < \mathit{Tlm}_1 > , \mathit{Nm} - < \mathit{Tlm}_2 > , O) = \\ \mathsf{reduce}(\mathsf{mark}(\mathit{Nm} - < \mathit{Tlm}_1 >), \mathit{Nm} - < \mathit{Tlm}_2 > , O) \\ \mid \mathsf{map\text{-}permutations}(\mathsf{permute}(\mathit{Tlm}_1), \mathit{Tlm}_2, O)\end{array}} \quad \text{[RRe4]}$$

The current top Node is the marked version of the goal top Node. The first option is to unmark it by applying rule *3* one or more times (one time for each current child being copied) in order to achieve the goal children list. For that purpose, get-permute provides map-permutations with all permutations of all possible concatenations of rule *3*.
The second option follows rule *4* by seeking a current child, that reduces to the goal Tree. This is done by oneleft2right.

$$\frac{\begin{array}{c}\mathsf{var}(\mathit{Nm}_1) = \mathsf{false}, \\ \mathsf{conv}(\mathit{Nm}_1) = \mathit{Nm}_2\end{array}}{\begin{array}{l}\mathsf{reduce}(\mathit{Nm}_1 - < \mathit{Tlm}_1 > , \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O) = \\ \mathsf{map\text{-}permutations}(\mathsf{get\text{-}permute}(\mathsf{number}(\mathit{Tlm}_2), \mathit{Tlm}_1), \mathit{Tlm}_2, O) \\ \mid \mathsf{oneleft2right}(\mathit{Tlm}_1, \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O)\end{array}} \quad \text{[RRe5]}$$

The current top Node is not marked and not equal to the goal top Node. Only rule *1* can be applied: mark the current top Node and proceed.

$$\frac{\begin{array}{r}\mathsf{var}(\mathit{Nm}_1) = \mathsf{false}, \\ \mathsf{var}(\mathit{Nm}_2) = \mathsf{false}, \\ \mathsf{is\text{-}marked}(\mathit{Nm}_1) = \mathsf{false}, \\ \mathit{Nm}_1 \neq \mathit{Nm}_2\end{array}}{\begin{array}{l}\mathsf{reduce}(\mathit{Nm}_1 - < \mathit{Tlm}_1 > , \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O) = \\ \mathsf{reduce}(\mathsf{mark}(\mathit{Nm}_1 - < \mathit{Tlm}_1 >), \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O)\end{array}} \quad \text{[RRe6]}$$

The current top Node is marked, but not equal to the goal top Node. If the goal top Node is lesser than the current top Node, rule *2* can be applied, and left2everyright tries to reduce a copy of the current Tree to every goal child.

In any case, we can apply rule *4* and let oneleft2right seek a current child, that reduces to the goal Tree.

$$\frac{\begin{array}{r}\mathsf{var}(\mathit{Nm}_1) = \mathsf{false},\\ \mathsf{var}(\mathit{Nm}_2) = \mathsf{false},\\ \mathsf{is\text{-}marked}(\mathit{Nm}_1) = \mathsf{true},\\ \mathsf{conv}(\mathit{Nm}_1) \neq \mathit{Nm}_2\end{array}}{\begin{array}{l}\mathsf{reduce}(\mathit{Nm}_1 - < \mathit{Tlm}_1 > , \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O) =\\ \mathsf{if}(\mathsf{gt}(\mathit{Nm}_1, \mathit{Nm}_2, O), \mathsf{left2everyright}(\mathit{Nm}_1 - < \mathit{Tlm}_1 > , \mathit{Tlm}_2, O))\\ \mid \mathsf{oneleft2right}(\mathit{Tlm}_1, \mathit{Nm}_2 - < \mathit{Tlm}_2 > , O)\end{array}} \quad \text{[RRe7]}$$

If successful, function reduce implements the $\rightsquigarrow^+$relation. This is easily verified:
Each of the Equations RRe4 to RRe7 performs one or more $\rightsquigarrow^+$-reduction steps.
Since we expect a successfully-parsed specification, the left-hand side (which is our current Tree at the start) can never be equal to the right-hand side (which is our goal Tree), so Equation RRe1 will not be the first applicable equation. At least one $\rightsquigarrow^+$-reduction step will have to be performed by one of the other equations.
If the left-hand side is a single variable, it should be equal to the right-hand side for a successful reduction, since variables cannot be marked and no $\rightsquigarrow$-rules apply. However, as stated above, the left-hand side and right-hand side are not equal. Therefore, Equation RRe2 can never be applied as the first option of a successful reduction. If Equation RRe3 is applied as a first option of a successful reduction, at least one $\rightsquigarrow^+$-reduction will be performed due to Proposition 7.6.

Function map reduces the children in the current list to the children in the goal list. This is done in pairs.

$$\mathsf{map}(, , O) \;=\; \mathsf{true} \quad \text{[RMa1]}$$

$$\begin{array}{l}\mathsf{map}(\mathit{Trm}_1 \mid \mathit{Tlm}_1, \mathit{Trm}_2 \mid \mathit{Tlm}_2, O) =\\ \mathsf{reduce}(\mathit{Trm}_1, \mathit{Trm}_2, O) \;\&\; \mathsf{map}(\mathit{Tlm}_1, \mathit{Tlm}_2, O)\end{array} \quad \text{[RMa2]}$$

$$\mathsf{map}(\mathit{Tlm}_1, \mathit{Tlm}_2, O) \;=\; \mathsf{false} \quad \textbf{otherwise} \quad \text{[RMa]}$$

Function map-permutations lets map try every Tree-List in the set of permutations. The lazy implementation of the Boolean '|' ensures that we stop as soon as a proper permutation is found.

$$\mathsf{map\text{-}permutations}(, \mathit{Tlm}, O) \;=\; \mathsf{false} \quad \text{[RPm1]}$$

$$\begin{array}{l}\mathsf{map\text{-}permutations}(\mathit{Tlm}^{+\,\hat{}\hat{}}\ \mathit{Pm}^*, \mathit{Tlm}, O) =\\ \mathsf{map}(\mathit{Tlm}^+, \mathit{Tlm}, O) \mid \mathsf{map\text{-}permutations}(\mathit{Pm}^*, \mathit{Tlm}, O)\end{array} \quad \text{[RPm2]}$$

Function left2everyright tries to reduce a copy of the current Tree to every Tree in the goal list.

$$\mathsf{left2everyright}(\mathit{Trm}, , O) \;=\; \mathsf{true} \quad \text{[RLe1]}$$

$\mathsf{left2everyright}(\mathit{Trm}_1,\ \mathit{Trm}_2|\ \mathit{Tlm},\ O) =$ [RLe2]
$\mathsf{reduce}(\mathit{Trm}_1,\ \mathit{Trm}_2,\ O)\ \&\ \mathsf{left2everyright}(\mathit{Trm}_1,\ \mathit{Tlm},\ O)$

Function oneleft2right tries to find a Tree in the current list that reduces to the goal Tree. The lazy implementation of the Boolean '|' ensures that we stop as soon as a proper Tree is found.

$\mathsf{oneleft2right}(,\ \mathit{Trm},\ O)\ =\ \mathsf{false}$ [ROl1]

$\mathsf{oneleft2right}(\mathit{Trm}_1|\ \mathit{Tlm},\ \mathit{Trm}_2,\ O) =$ [ROl2]
$\mathsf{reduce}(\mathit{Trm}_1,\ \mathit{Trm}_2,\ O)\ |\ \mathsf{oneleft2right}(\mathit{Tlm},\ \mathit{Trm}_2,\ O)$

7.2.6 Module RPO-Tools

Function embeddable determines whether the first Tree is embeddable in the second. To do this, the $\searrow$-reduction steps are performed on the second Tree and its permuted children, until the two Trees are equal, or it turns out to be impossible.

$\mathsf{embeddable}(\mathit{Tr},\ \mathit{Tr})\ =\ \mathsf{true}$ [REm1]

$\mathsf{embeddable}(\mathit{Tr},\ )\ =\ \mathsf{false}$ [REm2]

$$\frac{\mathit{Tl}_1 \neq \mathit{Tl}_2}{\begin{array}{l}\mathsf{embeddable}(N - < \mathit{Tl}_1 > ,\ N - < \mathit{Tl}_2 >) = \\ \mathsf{embed\text{-}permutations}(\mathit{Tl}_1,\ \mathsf{permute}(\mathit{Tl}_2))\ |\ \mathsf{embeddable}(N - < \mathit{Tl}_1 > ,\ \mathit{Tl}_2)\end{array}} \quad \text{[REm3]}$$

$\mathsf{embeddable}(\mathit{Tr}_1,\ \mathit{Tr}_2|\ \mathit{Tl}^{+}) =$ [REm4]
$\mathsf{embeddable}(\mathit{Tr}_1,\ \mathit{Tr}_2)\ |\ \mathsf{embeddable}(\mathit{Tr}_1,\ \mathit{Tl}^{+})$

$\mathsf{embeddable}(N_1 - < \mathit{Tl}_1 > ,\ N_2 - < \mathit{Tl}_2 >) =$ [REm]
$\mathsf{embeddable}(N_1 - < \mathit{Tl}_1 > ,\ \mathit{Tl}_2)$ **otherwise**

Function embed-list is needed to compare two Tree-Lists pairwise and find out if the pairs are embeddable.

$\mathsf{embed\text{-}list}(\mathit{Tl},\ \mathit{Tl})\ =\ \mathsf{true}$ [REl1]

$$\frac{\mathit{Tl}_1 \neq \mathit{Tl}_2}{\mathsf{embed\text{-}list}(\mathit{Tr}|\ \mathit{Tl}_1,\ \mathit{Tr}|\ \mathit{Tl}_2) = \mathsf{embed\text{-}list}(\mathit{Tl}_1,\ \mathit{Tl}_2)} \quad \text{[REl2]}$$

$$\frac{Tr_1 \neq Tr_2}{\begin{array}{l}\text{embed-list}(Tr_1|\ Tl_1,\ Tr_2|\ Tl_2) = \\ \text{embeddable}(Tr_1,\ Tr_2)\ \&\ \text{embed-list}(Tl_1,\ Tl_2)\end{array}} \quad \text{[REl3]}$$

$$\text{embed-list}(Tl_1,\ Tl_2) = \text{false} \quad \textbf{otherwise} \quad \text{[REl]}$$

Function embed-permutations is needed to find a successful permutation in which the first Tree-List is embeddable. The lazy implementation of the Boolean '|' ensures that we stop as soon as a proper permutation is found.

$$\text{embed-permutations}(Tl,\ ) = \text{false} \quad \text{[REp1]}$$

$$\begin{array}{l}\text{embed-permutations}(Tl,\ Tl^{+\,\wedge\wedge}\ P^*) = \\ \text{embed-list}(Tl,\ Tl^+)\ |\ \text{embed-permutations}(Tl,\ P^*)\end{array} \quad \text{[REp2]}$$

Auxiliary functions

Function get-permute implements a concatenation of rule *3*. It tries to make the first Tree-List match the second by getting al variations of deleting and copying arguments. At least one argument has to be deleted, copied more than once, or marked. When all legal variations have been found, they are permuted by permute.

7.2.7 Auxiliary modules

In Appendix D the auxiliary modules **Permutations**, **Order-Tools**, **Orders** and **SN-Booleans** are listed. Here, function permute and auxiliary types as Permutations and Orders are defined, as well as the composition of Boolean values and error messages and the necessary Boolean functions.

7.3 Thoughts on extensions

7.3.1 Extension of $\rightsquigarrow$-rule *3*

Currently, $\rightsquigarrow$-rule *3* enables us to make an arbitrary number of arguments below a non-variable node disappear, as long as the top node of the arguments is markable. This enables us even for functions with a fixed number of arguments to replace markable arguments with other (marked) arguments. But more could be wished for. Having a list construction in which double instances of the same element are replaced by one, a typical ASF+SDF equation could be:

$$O_1^* \#\ O \#\ O_2^* \#\ O \#\ O_3^* = O_1^* \#\ O \#\ O_2^* \#\ O_3^* \quad \text{[Or2]}$$

This equation will cause the RPO test to fail, because variables of the element sort may not be deleted using $\rightsquigarrow$-rule *3*, since they are not markable. Yet it is easy to see that the right-hand side is smaller in the RPO-sense than the left-hand side, and this is a type of ASF+SDF programming that is used frequently. The question of how to expand the $\rightsquigarrow$-rules and its Strong Normalization proof to meet this need remains unanswered.

7.3.2 Conditional specifications

Since our notion of CTRSs conforms to [DOS88], we could use their Proposition 4 to prove Strong Normalization. This proposition states that a conditional rewrite system is decreasing (and therefore SN) if there exists a well-founded extension $>$ of *searrow*, such that $>$ contains $\longrightarrow^+$ and $l^\sigma > l_i^\sigma, r_i^\sigma$ for each rule

$$\mathsf{lab}: \; l \longrightarrow r \text{ if } C_1 \& \cdots \& C_n \quad (C_i \;\equiv\; l_i == r_i \text{ or } C_i \;\equiv\; l_i <> r_i)$$

and substitution σ. We know our RPO contains $\longrightarrow^+$ and is an extension of *searrow*, so we could use RPO as a well-founded extension. That is, simply reuse that very Recursive Path Ordering for testing whether a left-hand side is also RPO-bigger than each side of each condition in the rule.

Since every side of every condition is rewritten to find out if it succeeds, we can try to establish orders for and $\leadsto^+$-reduce the left-hand side of the rule to every side of every condition. If this strategy succeeds, we certainly know our CTRS to be SN.

A problem is posed by positive conditions of which one side introduces a new variable:

$$\mathsf{lab} \;:\; l \longrightarrow r \text{ if } C_1, \ldots, C_n$$

and $C_i \;\equiv\; l_i == r_i$.
Without loss of generality we can suppose that l_i introduces a new variable x. In case of success of the condition, l_i receives value r_i. In case of failure this equation would not be applied, so we only have to consider the case of success in the following.
If $l_i \equiv x$ and $l \leadsto^+ r_i$ and l_i receives value r_i, then $l \leadsto^+ l_i \equiv x$.
If x is a subterm of l_i, we know that $l_i \leadsto^+ x$ because of Proposition 7.6. So if $l \leadsto^+ l_i \leadsto^+ x$, we can conclude that $l \leadsto^+ x$.

The strategy could then be to take the conditions one by one, $\leadsto^+$-reduce the left-hand side to every non-introducing side, and for every introducing side: store the knowledge that these new variables are RPO-smaller.
As soon as we encounter a side $t_i \equiv C[x]$, where x is such a variable, we know that $l \leadsto^+ x$ and we want to know $l \leadsto^+ C[x]$. We can never get $l \leadsto^+ C[l]$, because we would in general have to mark at least one node. But if we can get $l \leadsto^+ C[l']$ where $l \stackrel{1}{\leadsto} l'$ (only mark the top node of l), we could prove that $l' \leadsto^* x$, implying that $C[l'] \leadsto^* C[x]$. Then trivially $l \leadsto^+ C[l'] \leadsto^* C[x]$ yields $l \leadsto^+ C[x]$.

The first point is then to ensure that given such a $t_i \equiv C[x]$, $l \leadsto^+ C[l']$ (matter of implementation) and the next point, to prove that $l \leadsto^+ x$ and $l \stackrel{1}{\leadsto} l'$ imply that $l' \leadsto^* x$ (probably a rainy Sunday puzzle).

Undoubtedly this strategy would cover only a few of the types of frequently-used ASF+SDF specifications, but it certainly is a start.

Chapter 8

What to conclude from these properties

If your specification satisfies some of the implemented checks, more can be concluded. In this chapter we list the most important properties and implications. Proofs can be found in [Klo92]. Note that several properties used in the implications are not implemented. You will have to verify these for your specification by hand.

Definition 8.1 *A TRS or CTRS is* Weakly Normalizing (WN) *if for each $t \in \mathcal{T}(\Sigma)$ there is a rewrite sequence that ends in a normal form.*

Definition 8.2 *Let $s, t \in \mathcal{T}(\Sigma)$. We write $s = t$ if there is a finite sequence:*

$$s \equiv t_0 \longleftrightarrow t_1 \longleftrightarrow \ldots \longleftrightarrow t_n \equiv t (t_0, t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$$

where each $t_i \longleftrightarrow t_{i+1}$ stands for $t_i \longrightarrow t_{i+1}$ or $t_{i+1} \longrightarrow t_i$.

Definition 8.3 *A TRS or CTRS is* Weakly Church-Rosser (WCR) *if for every $s, t, u \in \mathcal{T}(\Sigma)$ there is a $v \in \mathcal{T}(\Sigma)$ such that:*

$$s \longrightarrow t \text{ and } s \longrightarrow u \Longrightarrow t \longrightarrow^* v \text{ and } u \longrightarrow^* v$$

Definition 8.4 *A TRS or CTRS is* Church-Rosser (CR) *if for every $s, t, u \in \mathcal{T}(\Sigma)$ there is a $v \in \mathcal{T}(\Sigma)$ such that:*

$$s \longrightarrow^* t \text{ and } s \longrightarrow^* u \Longrightarrow t \longrightarrow^* v \text{ and } u \longrightarrow^* v$$

Definition 8.5 *A TRS or CTRS has the* Unique Normal form property (UN) *if for every $s, t \in \mathcal{T}(\Sigma)$:*

$$s = t \text{ and } s, t \text{ are normal forms } \Longrightarrow s \equiv t$$

The following propositions state useful information in case a specifications satisfies the implemented checks:

Proposition 8.6 *If a TRS R is orthogonal and non-erasing, then: R is Weakly Normalizing $\Longleftrightarrow$ it is Strongly Normalizing.*

Proposition 8.7 *Every orthogonal TRS is CR.*

Proposition 8.8 *Every TRS that is a recursive program scheme, is orthogonal, hence CR.*

Definition 8.9

1. *Let R be a CTRS, and let P be an n-ary predicate on the terms $\in \mathcal{T}(\Sigma)$. Then P is* closed with respect to $\longrightarrow$ *if for all terms s_i, t_i such that $s_i \longrightarrow^* t_i$ $(i = 1, \ldots, n)$:*

$$P(s_1, \ldots, s_n) \Longrightarrow P(t_1, \ldots, t_n)$$

2. *Let R be a CTRS. Then R is* closed *if all conditions (appearing in some conditional rewrite rule of R), viewed as predicates with the variables ranging over the terms $\in \mathcal{T}(\Sigma)$ are closed with respect to $\longrightarrow$.*

Proposition 8.10 *An orthogonal, closed CTRS is confluent.*

The following theorem puts all properties together:

Theorem 8.11 *For any TRS or CTRS:*

1. *SN $\Longrightarrow$ WN*
2. *CR $\Longrightarrow$ WCR*
3. *CR $\Longrightarrow$ UN*
4. *SN and WCR $\Longrightarrow$ CR (Newman's Lemma)*
5. *UN and WN $\Longrightarrow$ CR*

The most powerful result that can be reached for any specification is completeness. A complete TRS will always yield the a normal form, and it will always yield equal normal forms for equal terms.

Definition 8.12 *A TRS or CTRS is* complete *or* canonical *if it is SN and CR. It is said to be a* decision procedure.

Chapter 9

Final remarks

9.1 Current state of things

Everybody interested in trying this tool on their own ASF+SDF specifications can get their own copy of the ASF+SDF specifications via `ftp` or `netscape` from

`http://www.cwi.nl/pub/gipe/AsFix/`

Get the file `TRS-analysis.tar.gz` and follow the instructions in `README`.

Suppose we have an ASF+SDF Meta-environment, running a specification consisting of one module `Module` in directory `dirname`, and we want to verify one of the implemented properties.

- Type `(Modules2AsFix)` in the LeLisp window. Module `Module` is translated to ASFIX. The result is file `Module.asfix` in directory `dirname`.
- Get a copy from the code as described above.
  If everything is unpacked without problems, open a new ASF+SDF Meta-environment in the newly-built directory `/AsFix-analysis/aterms`. Now all modules needed for the ASFIX format are read automatically.
- Open a term over sort ATerms.
  Type after Directory: `dirname` and after Filename: `Module.asfix`.
- Push the button for one of the implemented tests:
  - Non-Erasing
  - Orthogonal
  - Recursive Program Scheme
  - Strongly Normalizing
- Wait for the answer in the LeLisp window. The ASF+SDF Meta-environment will have to read all modules necessary for the test we selected, before it can perform this test. Be aware that the Strongly Normalizing test requires a long time, especially for large specifications.

The answer in the LeLisp window is a composition of a Boolean value and a message explaining the success or failure: *Bool* ↔ *Msg*. *Bool* indicates whether our specification does or does not satisfy this property. However, if we chose the test for Strongly Normalizing, the Boolean false means that the implemented Recursive Path Ordering cannot prove this for our specification. In that case we still do not know whether our specification is or is not Strongly Normalizing.

9.2 Future work

9.2.1 Future use of the tests

Obviously, the procedure described in Section 9.1 is not what we had in mind while doing this implementation in ASF+SDF. The goal is to have these tests as features in the new ASF+SDF Meta-environment, where we can pull the Specification menu, containing `Add`, `Save` and `Quit` options today, which will then be extended with a button for each of the tests. Hopefully, this implementation will also speed up the tests.

9.2.2 Changes in ASF+SDF

List constructions as output sort

Currently, it is not possible to specify context-free functions with a list construction as output sort. Plans exist to make this possible, which means a big change in the CTRS theory we use, and in some of the tests. We observe that variable-introducing conditions are not necessary anymore, since every variable-introducing condition can be replaced by a non-introducing condition using a few extra functions and keeping the same semantics. When this feature is forbidden, the way of specifying will become theoretically more sound.

The change will lead to the alteration of function fit-on within the Orthogonal test. We can think of two functions with the same name, of which one has a $*$ variant and the other the $+$ variant of the same list construction output sort. If this is allowed in the new ASF+SDF semantics, we will have to decide if two such functions can fit each other.

Open term rewriting

The only terms that are parsed and reduced successfully, are terms without variables. Since ASF+SDF forces us to make a proper distinction between the names of variables and other elements, the syntactical impact of this change would not be hard to grasp. The semantics of reducing would change tremendously.

The first problem we face is the introduction of variables in conditions. This will become impossible without some special notation, since facing a variable-introducing condition we will have to choose whether we match the variable-introducing side to the normal form of the other side, or try to rewrite the variable-introducing side to normal form and make a syntactical comparison. We can even have two variable-introducing sides within one condition.

In combination with the list construction change mentioned previously, the consideration above will become useless, if variable introduction is skipped whatsoever. Considerations related in Section 7.3.2 will change so drastically we cannot begin to think of a new set-up.

9.2.3 Extensions to test and theory

Forbidden features of ASF+SDF

As can be read in Section 3, the theory does not yet support every feature of ASF+SDF. We would like to see a sound theoretical basis for the use of default equations.

More properties

Not every property that is mentioned in Chapter 8 is implemented here. The properties *Weakly Normalizing*, *Weak Church Rosser* and *Unique Normal form* property can also tell a lot about a specification. The implementation of these three will involve actual reduction, so it is probably a better idea to implement this on the basis of the reduction strategies used in the new ASF+SDF Meta-environment.

We can also think of various other strategies to decide the undecidable SN, e.g. mapping terms to polynomial functions ([BKM89], Chapter 4).

Using external tools

The failure of our adventures with RRL does not imply the impossibility of using external tools for verification of properties. Certainly in the new ASF+SDF Meta-environment, it will be much easier to combine the Meta-environment with other applications.

We will need an adapter that transforms the AsFix format to the required input format for the external tool. Considering the level of this transforming operation, it might not be the best thing to try to implement this in a meta-level ASF+SDF specification.

It is also important to decide at an early stage up to what point the Meta-environment takes care of the communication with the external tool, and from where the user has to make decisions.

Appendix A

ASF+SDF specifications Non Erasing

A.1 Non-Erasing

imports Make-Rules$^{F.8}$ NE-Booleans$^{A.3}$ Compare$^{A.2}$
exports
  context-free syntax
    non-erasing(ATerm) $\rightarrow$ NE-BOOL
    non-erasing(Rules) $\rightarrow$ NE-BOOL
hiddens
  context-free syntax
    ne-check(Rules) $\rightarrow$ NE-BOOL
    check-conditions(Rules) $\rightarrow$ NE-BOOL
equations

$$\text{non-erasing}(T) \ = \ \text{non-erasing}(\text{asfix2tree}(T)) \qquad \text{[NNe1]}$$

$$\text{non-erasing}(R^*) \ = \ \text{if}(\text{check-conditions}(R^*), \text{ne-check}(R^*)) \qquad \text{[NNe2]}$$

$$\text{ne-check}() \ = \ \text{true} \leftrightarrow \text{Non-erasing} \qquad \text{[NNc1]}$$

$$\text{ne-check}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^* \ R^*) = \qquad \text{[NNc2]}$$
 if(compare(strip(Tr_1), strip(Tr_2)),
   ne-check(R^*),
   false $\leftrightarrow$ Erasing eq: L)

$$\text{check-conditions}() \ = \ \text{true} \leftrightarrow \text{Unconditional specification} \qquad \text{[NCc1]}$$

$$\text{check-conditions}(L : Tr_1 \longrightarrow Tr_2 \text{ if } \ R^*) = \qquad \text{[NCc2]}$$
 if(default(L),
   false $\leftrightarrow$ Not implemented for default-equations,
   check-conditions(R^*))

check-conditions(L : $Tr_1 \longrightarrow Tr_2$ if C^+ R^*) = [NCc3]
false $\leftrightarrow$ Not implemented for conditional specifications

A.2 Compare

imports Tree-Tools[F.7]
exports
context-free syntax
compare(Tree-List, Tree-List) $\to$ BOOL
strip(Tree-List) $\to$ Tree-List
equations

compare(, Tl)	=	true	[Cco1]
compare(N − < >| Tl_1 , Tl_2)	=	in(N, Tl_2) & compare(Tl_1, Tl_2)	[Cco2]

strip()	=		[CSt1]
strip((L, Nat, Tp) − < Tl >)	=	strip(Tl)	[CSt2]
strip((L, Var, Tp) − < >)	=	(L, Var, Tp) − < >	[CSt3]
strip((L, MetaVar, Tp) − < >)	=		[CSt4]
strip((L, Lex, Tp) − < Tl >)	=		[CSt5]
strip((L, ∗, Tp) − < Tl >)	=	strip(Tl)	[CSt6]
strip((L, +, Tp) − < Tl >)	=	strip(Tl)	[CSt7]

$$\frac{\text{strip}(Tr) = Tl_1,\ \text{strip}(Tl^+) = Tl_2}{\text{strip}(Tr|\ Tl^+) = Tl_1|\ Tl_2}$$ [CSt8]

A.3 NE-Booleans

imports Booleans[F.6] Literals[E.2]
exports
sorts NE-MSG NE-BOOL
context-free syntax

"Non-erasing"	$\to$ NE-MSG	
"Erasing" "eq:" Literal	$\to$ NE-MSG	
"Unconditional" "specification"	$\to$ NE-MSG	
"Not" "implemented" "for" "default-equations"	$\to$ NE-MSG	
"Not" "implemented" "for" "conditional" "specifications"	$\to$ NE-MSG	
BOOL "$\leftrightarrow$" NE-MSG	$\to$ NE-BOOL	
NE-BOOL "|" NE-BOOL	$\to$ NE-BOOL	{**left**}
if(BOOL, NE-BOOL, NE-BOOL)	$\to$ NE-BOOL	
if(NE-BOOL, NE-BOOL)	$\to$ NE-BOOL	

variables
$\mathit{Nmsg}\ [\mathit{0\text{-}9'}]* \rightarrow \mathsf{NE\text{-}MSG}$
$\mathit{Nbool}\ [\mathit{0\text{-}9'}]* \rightarrow \mathsf{NE\text{-}BOOL}$
equations
Boolean |-function

$$\mathsf{true} \leftrightarrow \mathit{Nmsg} \mid \mathit{Nbool} \ = \ \mathsf{true} \leftrightarrow \mathit{Nmsg} \qquad [\mathsf{NOr1}]$$
$$\mathsf{false} \leftrightarrow \mathit{Nmsg} \mid \mathit{Nbool} \ = \ \mathit{Nbool} \qquad [\mathsf{NOr2}]$$

Conditional functions:
If (first argument) then (second argument) else (third argument)

$$\mathsf{if}(\mathsf{true}, \mathit{Nbool}_1, \mathit{Nbool}_2) \ = \ \mathit{Nbool}_1 \qquad [\mathsf{NIf1}]$$
$$\mathsf{if}(\mathsf{false}, \mathit{Nbool}_1, \mathit{Nbool}_2) \ = \ \mathit{Nbool}_2 \qquad [\mathsf{NIf2}]$$

If (first argument) then (second argument) else (first argument)

$$\mathsf{if}(\mathsf{true} \leftrightarrow \mathit{Nmsg}, \mathit{Nbool}) \ = \ \mathit{Nbool} \qquad [\mathsf{NIf3}]$$
$$\mathsf{if}(\mathsf{false} \leftrightarrow \mathit{Nmsg}, \mathit{Nbool}) \ = \ \mathsf{false} \leftrightarrow \mathit{Nmsg} \qquad [\mathsf{NIf4}]$$

Appendix B

ASF+SDF specifications Orthogonal

B.1 Orthogonal

imports Left-Linear$^{B.2}$ Overlapping$^{B.3}$ Make-Rules$^{F.8}$
exports
  context-free syntax
    orthogonal(ATerm) $\rightarrow$ Orth-BOOL
    orthogonal(Rules) $\rightarrow$ Orth-BOOL
hiddens
  context-free syntax
    check-default(Rules) $\rightarrow$ Orth-BOOL
equations

orthogonal(T) = orthogonal(asfix2tree(T)) [OOr1]

orthogonal(R^*) = [OOr2]
 if(check-default(R^*),
  if(left-linear(R^*),
   if(invert(overlapping(R^*)), true $\leftrightarrow$ Left-linear and not overlapping)))

check-default() = true $\leftrightarrow$ No default-equations [OCc1]

check-default(L : $Tr_1 \longrightarrow Tr_2$ if C^* R^*) = [OCc2]
 if(default(L),
  false $\leftrightarrow$ Not implemented for specifications with default-equations,
  check-default(R^*))

B.2 Left-Linear

imports Tree-Tools$^{F.7}$ Orth-Booleans$^{B.5}$ Rules$^{F.1}$
exports
  context-free syntax
    left-linear(Rules) $\rightarrow$ Orth-BOOL
hiddens
  context-free syntax
    check-vars(Tree-List) $\rightarrow$ BOOL
    strip(Tree-List) $\rightarrow$ Tree-List
equations

$$\text{left-linear}() \;=\; \text{true} \leftrightarrow \text{Left-linear} \qquad \text{[LLl1]}$$

$$\text{left-linear}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^* \; R^*) = \qquad \text{[LLl2]}$$
$$\text{if}(\text{check-vars}(\text{strip}(Tr_1)), \text{left-linear}(R^*), \text{false} \leftrightarrow \text{Not left-linear eq: } L)$$

$$\text{check-vars}() \;=\; \text{true} \qquad \text{[LCv1]}$$
$$\text{check-vars}(N - < \; >|\; Tl) \;=\; \text{not}(\text{in}(N,\; Tl))\; \&\; \text{check-vars}(Tl) \qquad \text{[LCv2]}$$

$$\text{strip}() \;=\; \qquad \text{[LSt1]}$$
$$\text{strip}((L,\; Nat,\; Tp) - < Tl >) \;=\; \text{strip}(Tl) \qquad \text{[LSt2]}$$
$$\text{strip}((L,\; \text{Var},\; Tp) - < \; >) \;=\; (L,\; \text{Var},\; Tp) - < \; > \qquad \text{[LSt3]}$$
$$\text{strip}((L,\; \text{MetaVar},\; Tp) - < \; >) \;=\; \qquad \text{[LSt4]}$$
$$\text{strip}((L,\; \text{Lex},\; Tp) - < Tl >) \;=\; \qquad \text{[LSt5]}$$
$$\text{strip}((L,\; *,\; Tp) - < Tl >) \;=\; \text{strip}(Tl) \qquad \text{[LSt6]}$$
$$\text{strip}((L,\; +,\; Tp) - < Tl >) \;=\; \text{strip}(Tl) \qquad \text{[LSt7]}$$

$$\frac{\text{strip}(Tr) = Tl_1,\;\; \text{strip}(Tl^+) = Tl_2}{\text{strip}(Tr|\; Tl^+) = Tl_1|\; Tl_2} \qquad \text{[LSt8]}$$

B.3 Overlapping

imports Unifiable$^{B.4}$ Orth-Booleans$^{B.5}$ Rules$^{F.1}$
exports
  context-free syntax
    overlapping(Rules) $\rightarrow$ Orth-BOOL
hiddens
  context-free syntax
    another(Rules, Rules) $\rightarrow$ Orth-BOOL
    each(Rules, Rules) $\rightarrow$ Orth-BOOL
    itself(Rules) $\rightarrow$ Orth-BOOL
    Rules "minus" Rule $\rightarrow$ Rules

equations

$$\text{overlapping}(R^*) \; = \; \text{another}(R^*,\, R^*) \mid \text{itself}(R^*) \qquad \text{[Oov1]}$$

$$\text{another}(,\, R^*) \; = \; \text{false} \leftrightarrow \text{Not overlapping} \qquad \text{[OAn1]}$$

$$\text{another}(R\; R_1^*,\, R_2^*) \; = \; \text{each}(R,\, R_2^* \text{ minus } R) \mid \text{another}(R_1^*,\, R_2^*) \qquad \text{[OAn2]}$$

$$\text{each}(R,\,) \; = \; \text{false} \leftrightarrow \text{Not overlapping} \qquad \text{[OEa1]}$$

$$\text{each}(L_1 : \mathit{Tr}_1 \longrightarrow \mathit{Tr}_2 \text{ if } C_1^*,\, L_2 : \mathit{Tr}_3 \longrightarrow \mathit{Tr}_4 \text{ if } C_2^*\; R^*) = \qquad \text{[OEa2]}$$

$$\begin{array}{l} \text{if}(\text{unifiable}(\mathit{Tr}_1,\, \mathit{Tr}_3), \\ \quad \text{true} \leftrightarrow \text{Overlapping eq: } L_1 \text{ and eq: } L_2, \\ \quad \text{each}(L_1 : \mathit{Tr}_1 \longrightarrow \mathit{Tr}_2 \text{ if } C_1^*,\, R^*)) \end{array}$$

$$\text{itself}() \; = \; \text{false} \leftrightarrow \text{Not overlapping} \qquad \text{[OIt1]}$$

$$\text{itself}(L : \mathit{Tr}_1 \longrightarrow \mathit{Tr}_2 \text{ if } C^*\; R^*) = \qquad \text{[OIt2]}$$

$$\begin{array}{l} \text{if}(\text{unifiable}(\mathit{Tr}_1), \\ \quad \text{true} \leftrightarrow \text{Self-overlapping eq: } L, \\ \quad \text{itself}(R^*)) \end{array}$$

Remove a rule from a set of rules.

$$R_1^*\; R\; R_2^* \text{ minus } R \; = \; R_1^*\; R_2^* \qquad \text{[OMi1]}$$

$$R^* \text{ minus } R \; = \; R^* \quad \textbf{otherwise} \qquad \text{[OMi]}$$

B.4 Unifiable

imports Tree-Tools$^{F.7}$
exports
  context-free syntax
    unifiable(Tree) → BOOL
    unifiable(Tree, Tree-List) → BOOL
    fit-on(Tree-List, Tree-List) → BOOL
    fit-lists(Tree-List, Tree-List, Type) → BOOL
hiddens
  context-free syntax
    make-type(Node) → Type
    plus-type(Node) → Type
    star-type(Type) → Type
    star-type(Node) → Node
    star(Type) → BOOL
    star(Node) → BOOL

list-eq(Node, Node) $\rightarrow$ BOOL
list-eq(Type, Type) $\rightarrow$ BOOL
is-elt(Node, Type) $\rightarrow$ BOOL

equations

$$\mathrm{unifiable}(N - < Tl >) \ = \ \mathrm{unifiable}(N - < Tl > ,\ Tl) \qquad \text{[UUn1]}$$

$$\mathrm{unifiable}(N - < Tl > ,\ N - < Tl >) \ = \ \mathrm{not}(\mathrm{var}(N)) \qquad \text{[UUn2]}$$

$$\frac{Tl_1 \neq Tl_2,\quad \mathrm{func}(N) = \mathrm{true}}{\begin{array}{l}\mathrm{unifiable}(N - < Tl_1 > ,\ N - < Tl_2 >) = \\ \text{fit-on}(Tl_1,\ Tl_2) \mid \mathrm{unifiable}(N - < Tl_1 > ,\ Tl_2)\end{array}} \qquad \text{[UUn3]}$$

$$\frac{Tl_1 \neq Tl_2,\quad \mathrm{list}(N) = \mathrm{true}}{\begin{array}{l}\mathrm{unifiable}(N - < Tl_1 > ,\ N - < Tl_2 >) = \\ \text{fit-lists}(Tl_1,\ Tl_2,\ \text{make-type}(N)) \mid \mathrm{unifiable}(N - < Tl_1 > ,\ Tl_2)\end{array}} \qquad \text{[UUn4]}$$

$$\frac{N_1 \neq N_2,\quad \mathrm{not}(\mathrm{var}(N_1))\ \&\ (\mathrm{func}(N_2) \mid \mathrm{list}(N_2)) = \mathrm{true}}{\mathrm{unifiable}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 >) = \mathrm{unifiable}(N_1 - < Tl_1 > ,\ Tl_2)} \qquad \text{[UUn5]}$$

$$\mathrm{unifiable}(Tr_1,\ Tr_2 \mid Tl^{+}) \ = \ \mathrm{unifiable}(Tr_1,\ Tr_2) \mid \mathrm{unifiable}(Tr_1,\ Tl^{+}) \qquad \text{[UUn6]}$$

$$\mathrm{unifiable}(Tr,\ Tl) \ = \ \mathrm{false} \quad \textbf{otherwise} \qquad \text{[UUn]}$$

$$\text{fit-on}(,\ ) \ = \ \mathrm{true} \qquad \text{[UFo1]}$$

$$\frac{N_1 \neq N_2,\quad \mathrm{type}(N_1) = \mathrm{type}(N_2),\quad \mathrm{var}(N_1) \mid \mathrm{var}(N_2) = \mathrm{true}}{\begin{array}{l}\text{fit-on}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 >) = \\ \mathrm{var}(N_1)\ \&\ \mathrm{not}(\mathrm{list}(N_2)) \mid \mathrm{var}(N_2)\ \&\ \mathrm{not}(\mathrm{list}(N_1))\end{array}} \qquad \text{[UFo2]}$$

$$\frac{\mathrm{list}(N) = \mathrm{false}}{\text{fit-on}(N - < Tl_1 > ,\ N - < Tl_2 >) = \text{fit-on}(Tl_1,\ Tl_2)} \qquad \text{[UFo3]}$$

$$\frac{\mathrm{list}(N) = \mathrm{true}}{\text{fit-on}(N - < Tl_1 > ,\ N - < Tl_2 >) = \text{fit-lists}(Tl_1,\ Tl_2,\ \mathrm{type}(N))} \qquad \text{[UFo4]}$$

$$\frac{\begin{array}{c} N_1 \neq N_2, \\ \text{list-eq}(N_1,\ N_2) = \text{true} \end{array}}{\text{fit-on}(N_1 - <\ Tl_1 >\ ,\ N_2 - <\ Tl_2 >) = \text{fit-lists}(Tl_1,\ Tl_2,\ \text{plus-type}(N_1))} \quad \text{[UFo4]}$$

$$\text{fit-on}(Tr_1|\ Tl_1^+,\ Tr_2|\ Tl_2^+) \ = \ \text{fit-on}(Tr_1,\ Tr_2)\ \&\ \text{fit-on}(Tl_1^+,\ Tl_2^+) \quad \text{[UFo5]}$$

$$\text{fit-on}(Tl_1,\ Tl_2) \ = \ \text{false} \quad \textbf{otherwise} \quad \text{[UFo]}$$

$$\text{fit-lists}(,\ ,\ Tp) \ = \ \text{star}(Tp) \quad \text{[UFl1]}$$

$$\frac{\begin{array}{c} \text{var}(N)\ \&\ \text{star}(Tp) = \text{true}, \\ \text{type}(N) = Tp \end{array}}{\text{fit-lists}(,\ N - <\ >|\ Tl_2\ ,\ Tp) = \text{fit-lists}(,\ Tl_2,\ Tp)} \quad \text{[UFl2]}$$

$$\frac{\begin{array}{c} \text{var}(N)\ \&\ \text{star}(Tp) = \text{true}, \\ \text{type}(N) = Tp \end{array}}{\text{fit-lists}(N - <\ >|\ Tl_2\ ,\ ,\ Tp) = \text{fit-lists}(Tl_2,\ ,\ Tp)} \quad \text{[UFl3]}$$

$$\frac{\text{is-elt}(N_1,\ Tp)\ \&\ \text{is-elt}(N_2,\ Tp) = \text{true}}{\begin{array}{l} \text{fit-lists}(N_1 - <\ Tl_1 >|\ Tl_2\ ,\ N_2 - <\ Tl_3 >|\ Tl_4\ ,\ Tp) = \\ \text{fit-on}(N_1 - <\ Tl_1 >\ ,\ N_2 - <\ Tl_3 >)\ \&\ \text{fit-lists}(Tl_2,\ Tl_4,\ \text{star-type}(Tp)) \end{array}} \quad \text{[UFl4]}$$

$$\frac{\begin{array}{c} \text{not}(\text{var}(N_1))\ \&\ \text{not}(\text{var}(N_2)) = \text{true}, \\ \text{list-eq}(\text{type}(N_1),\ Tp)\ \&\ \text{list-eq}(\text{type}(N_2),\ Tp) = \text{true} \end{array}}{\begin{array}{l} \text{fit-lists}(N_1 - <\ Tl_1 >|\ Tl_2\ ,\ N_2 - <\ Tl_3 >|\ Tl_4\ ,\ Tp) = \\ \text{fit-on}(N_1 - <\ Tl_1 >\ ,\ N_2 - <\ Tl_3 >)\ \&\ \text{fit-lists}(Tl_2,\ Tl_4,\ \text{star-type}(Tp)) \end{array}} \quad \text{[UFl5]}$$

$$\frac{\begin{array}{c} \text{var}(N_2)\ \&\ \text{list-eq}(\text{type}(N_2),\ Tp)\ \&\ \text{star}(\text{type}(N_2)) = \text{true}, \\ \text{is-elt}(N_1,\ Tp)\ |\ \text{list-eq}(\text{type}(N_1),\ Tp)\ \&\ \text{not}(\text{var}(N_1)) = \text{true} \end{array}}{\begin{array}{l} \text{fit-lists}(N_1 - <\ Tl_1 >|\ Tl_2\ ,\ N_2 - <\ >|\ Tl_3\ ,\ Tp) = \\ \text{fit-lists}(N_1 - <\ Tl_1 >|\ Tl_2\ ,\ Tl_3,\ Tp) \\ |\ \text{fit-lists}(Tl_2,\ Tl_3,\ \text{star-type}(Tp)) \\ |\ \text{fit-lists}(Tl_2,\ N_2 - <\ >|\ Tl_3\ ,\ \text{star-type}(Tp)) \end{array}} \quad \text{[UFl6]}$$

$$\frac{\begin{array}{c} \text{var}(N_1)\ \&\ \text{list-eq}(\text{type}(N_1),\ Tp)\ \&\ \text{star}(\text{type}(N_1)) = \text{true}, \\ \text{is-elt}(N_2,\ Tp)\ |\ \text{list-eq}(\text{type}(N_2),\ Tp)\ \&\ \text{not}(\text{var}(N_2)) = \text{true} \end{array}}{\begin{array}{l} \text{fit-lists}(N_1 - <\ >|\ Tl_1\ ,\ N_2 - <\ Tl_2 >|\ Tl_3\ ,\ Tp) = \\ \text{fit-lists}(Tl_1,\ N_2 - <\ Tl_2 >|\ Tl_3\ ,\ Tp) \\ |\ \text{fit-lists}(Tl_1,\ Tl_3,\ \text{star-type}(Tp)) \\ |\ \text{fit-lists}(N_1 - <\ >|\ Tl_1\ ,\ Tl_3,\ \text{star-type}(Tp)) \end{array}} \quad \text{[UFl7]}$$

$$\frac{\begin{array}{c} \text{var}(N_2)\ \&\ \text{list-eq}(\text{type}(N_2),\ Tp)\ \&\ \text{not}(\text{star}(\text{type}(N_2))) = \text{true}, \\ \text{is-elt}(N_1,\ Tp)\ |\ \text{list-eq}(\text{type}(N_1),\ Tp)\ \&\ \text{not}(\text{var}(N_1)) = \text{true} \end{array}}{\begin{array}{l} \text{fit-lists}(N_1 - <\ Tl_1 >|\ Tl_2\ ,\ N_2 - <\ >|\ Tl_3\ ,\ Tp) = \\ \text{fit-lists}(Tl_2,\ Tl_3,\ \text{star-type}(Tp)) \\ |\ \text{fit-lists}(Tl_2,\ \text{star-type}(N_2) - <\ >|\ Tl_3\ ,\ \text{star-type}(Tp)) \end{array}} \quad \text{[UFl8]}$$

$$\frac{\begin{array}{c}\mathsf{var}(N_1)\ \&\ \mathsf{list\text{-}eq}(\mathsf{type}(N_1),\ Tp)\ \&\ \mathsf{not}(\mathsf{star}(\mathsf{type}(N_1))) = \mathsf{true},\\ \mathsf{is\text{-}elt}(N_2,\ Tp)\ |\ \mathsf{list\text{-}eq}(\mathsf{type}(N_2),\ Tp)\ \&\ \mathsf{not}(\mathsf{var}(N_2)) = \mathsf{true}\end{array}}{\begin{array}{l}\mathsf{fit\text{-}lists}(N_1 - <\ >|\ Tl_1\ ,\ N_2 - <\ Tl_2 >|\ Tl_3\ ,\ Tp) =\\ \mathsf{fit\text{-}lists}(Tl_1,\ Tl_3,\ \mathsf{star\text{-}type}(Tp))\\ |\ \mathsf{fit\text{-}lists}(\mathsf{star\text{-}type}(N_1) - <\ >|\ Tl_1\ ,\ Tl_3,\ \mathsf{star\text{-}type}(Tp))\end{array}} \quad [\mathsf{UFl9}]$$

$$\frac{\begin{array}{c}\mathsf{var}(N_1)\ \&\ \mathsf{var}(N_2) = \mathsf{true},\\ \mathsf{list\text{-}eq}(\mathsf{type}(N_1),\ Tp)\ \&\ \mathsf{list\text{-}eq}(\mathsf{type}(N_2),\ Tp) = \mathsf{true}\end{array}}{\begin{array}{l}\mathsf{fit\text{-}lists}(N_1 - <\ >|\ Tl_1\ ,\ N_2 - <\ >|\ Tl_2\ ,\ Tp) =\\ \mathsf{fit\text{-}lists}(Tl_1,\ Tl_2,\ \mathsf{star\text{-}type}(Tp))\\ |\ \mathsf{fit\text{-}lists}(\mathsf{star\text{-}type}(N_1) - <\ >|\ Tl_1\ ,\ Tl_2,\ \mathsf{star\text{-}type}(Tp))\\ |\ \mathsf{fit\text{-}lists}(Tl_1,\ \mathsf{star\text{-}type}(N_2) - <\ >|\ Tl_2\ ,\ \mathsf{star\text{-}type}(Tp))\end{array}} \quad [\mathsf{UFl10}]$$

$$\mathsf{fit\text{-}lists}(Tl_1,\ Tl_2,\ Tp)\ =\ \mathsf{false}\quad \textbf{otherwise} \qquad [\mathsf{UFl}]$$

Construct a list-type with a '+'.

$$\mathsf{plus\text{-}type}((L_1,*,\ L_2\ Sp\ *))\ =\ L_2\ L_1\ + \qquad [\mathsf{UPt1}]$$
$$\mathsf{plus\text{-}type}((L_1,+,\ L_2\ Sp\ +))\ =\ L_2\ L_1\ + \qquad [\mathsf{UPt1}]$$
$$\mathsf{plus\text{-}type}((L,\ A,\ Tp))\ =\ Tp\quad \textbf{otherwise} \qquad [\mathsf{UPt}]$$

Transform a list-type with a '+' to the same type with a '∗'.

$$\mathsf{star\text{-}type}(L\ Sp\ +)\ =\ L\ Sp\ * \qquad [\mathsf{USt1}]$$
$$\mathsf{star\text{-}type}((L,\ A,\ Tp))\ =\ (L,\ A,\ \mathsf{star\text{-}type}(Tp)) \qquad [\mathsf{USt2}]$$
$$\mathsf{star\text{-}type}(Tp)\ =\ Tp\quad \textbf{otherwise} \qquad [\mathsf{USt}]$$

$$\mathsf{star}(L\ Sp\ *)\ =\ \mathsf{true} \qquad [\mathsf{UStar1}]$$
$$\mathsf{star}((L,\ A,\ Tp))\ =\ \mathsf{star}(Tp) \qquad [\mathsf{UStar2}]$$
$$\mathsf{star}(Tp)\ =\ \mathsf{false}\quad \textbf{otherwise} \qquad [\mathsf{Star}]$$

Check whether these nodes represent '+' and '∗'-variants of the same list-construction.

$$\mathsf{list\text{-}eq}((L_1,*,\ L_2\ L_1\ *),(L_1,+,\ L_2\ L_1\ +))\ =\ \mathsf{true} \qquad [\mathsf{ULe1}]$$
$$\mathsf{list\text{-}eq}((L_1,+,\ L_2\ L_1\ +),(L_1,*,\ L_2\ L_1\ *))\ =\ \mathsf{true} \qquad [\mathsf{ULe1}]$$
$$\mathsf{list\text{-}eq}(N_1,\ N_2)\ =\ \mathsf{false}\quad \textbf{otherwise} \qquad [\mathsf{ULe}]$$

Check whether the first type is a '+'-variant of the second.

$$\mathsf{list\text{-}eq}(L\ Sp\ It_1,\ L\ Sp\ It_2)\ =\ \mathsf{true} \qquad [\mathsf{UPl1}]$$
$$\mathsf{list\text{-}eq}(Tp_1,\ Tp_2)\ =\ \mathsf{false}\quad \textbf{otherwise} \qquad [\mathsf{UPl}]$$

Determine whether the node is an element of the list-type.

$$\mathsf{is\text{-}elt}((L_1,\ A,\ L),\ L\ Sp\ *)\ =\ \mathsf{true} \qquad [\mathsf{Ule1}]$$
$$\mathsf{is\text{-}elt}((L_1,\ A,\ L),\ L\ Sp\ +)\ =\ \mathsf{true} \qquad [\mathsf{Ule1}]$$
$$\mathsf{is\text{-}elt}(N,\ Tp)\ =\ \mathsf{false}\quad \textbf{otherwise} \qquad [\mathsf{Ule}]$$

B.5 Orth-Booleans

imports Booleans[F.6] Literals[E.2]
exports
sorts Orth-MSG Orth-BOOL
context-free syntax

"Left-linear"	$\rightarrow$ Orth-MSG
"Not" "left-linear" "eq:" Literal	$\rightarrow$ Orth-MSG
"Not" "overlapping"	$\rightarrow$ Orth-MSG
"Self-overlapping" "eq:" Literal	$\rightarrow$ Orth-MSG
"Overlapping" "eq:" Literal "and" "eq:" Literal	$\rightarrow$ Orth-MSG
"Left-linear" "and" "not" "overlapping"	$\rightarrow$ Orth-MSG
"No" "default-equations"	$\rightarrow$ Orth-MSG
"Not" "implemented" "for" "specifications" "with" "default-equations"	$\rightarrow$ Orth-MSG
BOOL "$\leftrightarrow$" Orth-MSG	$\rightarrow$ Orth-BOOL
Orth-BOOL "|" Orth-BOOL	$\rightarrow$ Orth-BOOL {**left**}
invert(Orth-BOOL)	$\rightarrow$ Orth-BOOL
if(BOOL, Orth-BOOL, Orth-BOOL)	$\rightarrow$ Orth-BOOL
if(Orth-BOOL, Orth-BOOL)	$\rightarrow$ Orth-BOOL

variables
Omsg [*0-9'*]$*$ $\rightarrow$ Orth-MSG
Obool [*0-9'*]$*$ $\rightarrow$ Orth-BOOL
equations

Boolean or-function

$$\text{true} \leftrightarrow \mathit{Omsg} \mid \mathit{Obool} \;=\; \text{true} \leftrightarrow \mathit{Omsg} \tag{OOr1}$$
$$\text{false} \leftrightarrow \mathit{Omsg} \mid \mathit{Obool} \;=\; \mathit{Obool} \tag{OOr2}$$

Boolean negation-function

$$\text{invert}(\text{true} \leftrightarrow \mathit{Omsg}) \;=\; \text{false} \leftrightarrow \mathit{Omsg} \tag{OIv1}$$
$$\text{invert}(\text{false} \leftrightarrow \mathit{Omsg}) \;=\; \text{true} \leftrightarrow \mathit{Omsg} \tag{OIv2}$$

Conditional functions:
If (first argument) then (second argument) else (third argument)

$$\text{if}(\text{true}, \mathit{Obool}_1, \mathit{Obool}_2) \;=\; \mathit{Obool}_1 \tag{OIf1}$$
$$\text{if}(\text{false}, \mathit{Obool}_1, \mathit{Obool}_2) \;=\; \mathit{Obool}_2 \tag{OIf2}$$

If (first argument) then (second argument) else (first argument)

$$\text{if}(\text{true} \leftrightarrow \mathit{Omsg}, \mathit{Obool}) \;=\; \mathit{Obool} \tag{OIf3}$$
$$\text{if}(\text{false} \leftrightarrow \mathit{Omsg}, \mathit{Obool}) \;=\; \text{false} \leftrightarrow \mathit{Omsg} \tag{OIf4}$$

Appendix C

ASF+SDF specifications Recursive Program Scheme

C.1 RPS

imports RPS-Check$^{C.2}$ RPS-Booleans$^{C.3}$ Make-Rules$^{F.8}$
exports
  context-free syntax
    rps(ATerm) → RPS-BOOL
    rps(Rules) → RPS-BOOL
hiddens
  context-free syntax
    check-conditions(Rules) → RPS-BOOL
equations

rps(T) = rps(asfix2tree(T)) [RRp1]

rps(R^*) = if(check-conditions(R^*), rps-check(R^*,)) [RRp2]

check-conditions() = true ↔ Unconditional specification [RCc1]

check-conditions(L : Tr_1 ⟶ Tr_2 if R^*) = [RCc2]
if(default(L),
  false ↔ Not implemented for default-equations,
  check-conditions(R^*))

check-conditions(L : Tr_1 ⟶ Tr_2 if C^+ R^*) = [RCc3]
false ↔ Not implemented for conditional specifications

C.2 RPS-Check

imports RPS-Booleans$^{C.3}$ Rules$^{F.1}$ Tree-Tools$^{F.7}$
exports
  context-free syntax
    rps-check(Rules, Tree-List) → RPS-BOOL
hiddens
  context-free syntax
    check-head(Node) → BOOL
    check-func(Tree-List) → BOOL
    check-vars(Tree-List) → BOOL
    if(BOOL, BOOL, BOOL) → BOOL
equations

rps-check(, *Tl*) = [RRc1]
 if(check-func(*Tl*),
   true ↔ The unknown functions are: *Tl*,
   false ↔ More than one equation for unknown functions)

rps-check(L : $N - < Tl_1 > \longrightarrow Tr$ if C^* R^*, Tl_2) = [RRc2]
 if(check-head(N), if(check-vars(Tl_1), rps-check(R^*, Tl_2| $N - < >$),
   false ↔ Wrong argument list in lhs eq: L),
  false ↔ No functional lhs in eq: L)

check-head(N) = not(var(N)) [RCh1]

check-func() = true [RCf1]
check-func($N - < >$| *Tl*) = not(in(N, *Tl*)) & check-func(*Tl*) [RCf2]

check-vars() = true [RCv1]
check-vars($N - < >$| *Tl*) = var(N) & not(in(N, *Tl*)) & check-vars(*Tl*) [RCv2]
check-vars(*Tl*) = false **otherwise** [RCv]

C.3 RPS-Booleans

imports Booleans$^{F.6}$ Tree$^{F.2}$ Literals$^{E.2}$
exports
  sorts RPS-MSG RPS-BOOL
  context-free syntax
    "The" "unknown" "functions" "are:" Tree-List → RPS-MSG
    "More" "than" "one" "equation" "for" "unknown" "functions" → RPS-MSG
    "Wrong" "argument" "list" "in" "lhs" "eq:" Literal → RPS-MSG
    "No" "functional" "lhs" "in" "eq:" Literal → RPS-MSG

"Unconditional" "specification"	→ RPS-MSG
"Not" "implemented" "for" "default-equations"	→ RPS-MSG
"Not" "implemented" "for" "conditional" "specifications"	→ RPS-MSG
BOOL "↔" RPS-MSG	→ RPS-BOOL
RPS-BOOL "|" RPS-BOOL	→ RPS-BOOL {**left**}
if(BOOL, RPS-BOOL, RPS-BOOL)	→ RPS-BOOL
if(RPS-BOOL, RPS-BOOL)	→ RPS-BOOL

variables

Rmsg [*0-9'*]∗ → RPS-MSG

Rbool [*0-9'*]∗ → RPS-BOOL

equations

Boolean |-function

$$\text{true} \leftrightarrow \mathit{Rmsg} \mid \mathit{Rbool} \;=\; \text{true} \leftrightarrow \mathit{Rmsg} \qquad \text{[ROr1]}$$

$$\text{false} \leftrightarrow \mathit{Rmsg} \mid \mathit{Rbool} \;=\; \mathit{Rbool} \qquad \text{[ROr2]}$$

Conditional functions:
If (first argument) then (second argument) else (third argument)

$$\text{if}(\text{true}, \mathit{Rbool}_1, \mathit{Rbool}_2) \;=\; \mathit{Rbool}_1 \qquad \text{[RIf1]}$$

$$\text{if}(\text{false}, \mathit{Rbool}_1, \mathit{Rbool}_2) \;=\; \mathit{Rbool}_2 \qquad \text{[RIf2]}$$

If (first argument) then (second argument) else (first argument)

$$\text{if}(\text{true} \leftrightarrow \mathit{Rmsg}, \mathit{Rbool}) \;=\; \mathit{Rbool} \qquad \text{[RIf3]}$$

$$\text{if}(\text{false} \leftrightarrow \mathit{Rmsg}, \mathit{Rbool}) \;=\; \text{false} \leftrightarrow \mathit{Rmsg} \qquad \text{[RIf4]}$$

Appendix D

ASF+SDF specifications Strong Normalization

D.1 SN

imports Make-Rules$^{F.8}$ Get-Orders$^{D.3}$ RPO-Check$^{D.2}$
exports
  context-free syntax
    sn(ATerm) → SN-BOOL
    sn(Rules) → SN-BOOL
hiddens
  context-free syntax
    check-conditions(Rules) → SN-BOOL
equations

$$\mathsf{sn}(T) = \mathsf{sn}(\mathsf{asfix2tree}(T)) \quad \text{[SSn1]}$$

$$\mathsf{sn}(R^*) = \mathsf{if}(\mathsf{check\text{-}conditions}(R^*), \mathsf{rpo\text{-}check}(R^*, \mathsf{get\text{-}orders}(R^*, \{\}))) \quad \text{[SSn2]}$$

$$\mathsf{check\text{-}conditions}() = \mathsf{true} \leftrightarrow \text{Unconditional specification} \quad \text{[SCc1]}$$

$$\mathsf{check\text{-}conditions}(L : Tr_1 \longrightarrow Tr_2 \text{ if } R^*) = \mathsf{if}(\mathsf{default}(L), \mathsf{false} \leftrightarrow \text{Not implemented for default-equations}, \mathsf{check\text{-}conditions}(R^*)) \quad \text{[SCc2]}$$

$$\mathsf{check\text{-}conditions}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^+\ R^*) = \mathsf{false} \leftrightarrow \text{Not implemented for conditional specifications} \quad \text{[SCc3]}$$

D.2 RPO-Check

imports RPO-Reduce$^{D.5}$ Rules$^{F.1}$
exports
context-free syntax
rpo-check(Rules, Orders) $\rightarrow$ SN-BOOL
check-per-order(Rules, Order) $\rightarrow$ SN-BOOL
equations

rpo-check(R^*,) $=$ false $\leftrightarrow$ All orders failed [RCh1]

rpo-check(R^*, $O\#$ O^*) $=$ check-per-order(R^*, O) | rpo-check(R^*, O^*) [RCh2]

check-per-order(, O) $=$ true $\leftrightarrow$ All equations checked using order : O [RCp1]

check-per-order(L : Tr_1 $\longrightarrow$ Tr_2 if C^* R^*, O) $=$ [RCp2]
if(reduce(tree2mtree(Tr_1), tree2mtree(Tr_2), O),
check-per-order(R^*, O),
false $\leftrightarrow$ Failed reducing eq : L)

D.3 Get-Orders

imports Find-Relations$^{D.4}$ Rules$^{F.1}$
exports
context-free syntax
get-orders(Rules, Orders) $\rightarrow$ Orders
equations

get-orders(, O^+) $=$ O^+ [GGo1]

get-orders(R^*,) $=$ [GGo2]

$$\frac{\text{embeddable}(Tr_2,\ Tr_1) = \text{true}}{\text{get-orders}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^*\ R^*,\ O^+) = \text{get-orders}(R^*,\ O^+)}$$ [GGo3]

$$\frac{\text{embeddable}(Tr_2,\ Tr_1) = \text{false},\quad \text{find-relations}(Tr_1,\ Tr_2,\ O^+) = O^*}{\text{get-orders}(L : Tr_1 \longrightarrow Tr_2 \text{ if } C^*\ R^*,\ O^+) = \text{get-orders}(R^*,\ O^*)}$$ [GGo4]

D.4 Find-Relations

imports RPO-Tools[D.6] Order-Tools[H.2]
exports
context-free syntax
find-relations(Tree, Tree, Orders) → Orders
find-left(Tree-List, Tree, Orders) → Orders
every-right(Tree, Tree-List, Orders) → Orders
find-all-lefts(Tree-List, Tree-List, Orders) → Orders
equations

$$\frac{\text{embeddable}(Tr_2,\ Tr_1) = \text{true}}{\text{find-relations}(Tr_1,\ Tr_2,\ O^+) = O^+} \qquad \text{[FFr1]}$$

$$\frac{\begin{array}{r}\text{embeddable}(N - < Tl_2 > ,\ N - < Tl_1 >) = \text{false},\\ Tl_1 \neq Tl_2,\\ \text{find-all-lefts}(Tl_1,\ Tl_2,\ O^+) = O_1^*,\\ \text{find-left}(Tl_1,\ N - < Tl_2 > ,\ O^+) = O_2^*\end{array}}{\text{find-relations}(N - < Tl_1 > ,\ N - < Tl_2 > ,\ O^+) = O_1^* \#\ O_2^*} \qquad \text{[FFr2]}$$

$$\frac{\begin{array}{r}\text{embeddable}(N_2 - < Tl_2 > ,\ N_1 - < Tl_1 >) = \text{false},\\ N_1 \neq N_2,\\ \text{var}(N_1) = \text{false},\\ \text{var}(N_2) = \text{false},\\ \text{in}(N_2 > N_1,\ O) = \text{true},\\ \text{find-left}(Tl_1,\ N_2 - < Tl_2 > ,\ O) = O_1^*,\\ \text{find-relations}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 > ,\ O^*) = O_2^*\end{array}}{\text{find-relations}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 > ,\ O\#\ O^*) = O_1^* \#\ O_2^*} \qquad \text{[FFr3]}$$

$$\frac{\begin{array}{r}\text{embeddable}(N_2 - < Tl_2 > ,\ N_1 - < Tl_1 >) = \text{false},\\ N_1 \neq N_2,\\ \text{var}(N_1) = \text{false},\\ \text{var}(N_2) = \text{false},\\ \text{in}(N_2 > N_1,\ O) = \text{false},\\ \text{find-left}(Tl_1,\ N_2 - < Tl_2 > ,\ O) = O_1^*,\\ \text{every-right}(N_1 - < Tl_1 > ,\ Tl_2,\ \text{trans-close}(\text{add}(N_1 > N_2,\ O))) = O_2^*,\\ \text{find-relations}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 > ,\ O^*) = O_3^*\end{array}}{\text{find-relations}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 > ,\ O\#\ O^*) = O_1^* \#\ O_2^* \#\ O_3^*} \qquad \text{[FFr4]}$$

$\text{find-relations}(Tr_1,\ Tr_2,\ O^*)\ =$ **otherwise** [FFr]

$$\frac{\begin{array}{r}\text{find-relations}(Tr_1,\ Tr_2,\ O^+) = O_1^*,\\ \text{find-left}(Tl,\ Tr_2,\ O^+) = O_2^*\end{array}}{\text{find-left}(Tr_1|\ Tl,\ Tr_2,\ O^+) = O_1^* \#\ O_2^*} \qquad \text{[FFl1]}$$

$\text{find-left}(\mathit{Tl},\ \mathit{Tr},\ O^*) \ = \quad \textbf{otherwise}$ [FFl]

$\text{every-right}(\mathit{Tr}_1,\ ,\ O^+) \ = \ O^+$ [FEr1]

$$\frac{\text{find-relations}(\mathit{Tr}_1,\ \mathit{Tr}_2,\ O^+) = {O^+}'}{\text{every-right}(\mathit{Tr}_1,\ \mathit{Tr}_2|\ \mathit{Tl},\ O^+) = \text{every-right}(\mathit{Tr}_1,\ \mathit{Tl},\ {O^+}')}$$ [FEr2]

$\text{every-right}(\mathit{Tr},\ \mathit{Tl},\ O^*) \ = \quad \textbf{otherwise}$ [FEr]

$\text{find-all-lefts}(\mathit{Tl},\ ,\ O^+) \ = \ O^+$ [FMa1]

$$\frac{\text{find-left}(\mathit{Tl}^+,\ \mathit{Tr},\ O^+) = {O^+}'}{\text{find-all-lefts}(\mathit{Tl}^+,\ \mathit{Tr}|\ \mathit{Tl},\ O^+) = \text{find-all-lefts}(\mathit{Tl}^+,\ \mathit{Tl},\ {O^+}')}$$ [FMa2]

$\text{find-all-lefts}(\mathit{Tl}_1,\ \mathit{Tl}_2,\ O^*) \ = \quad \textbf{otherwise}$ [FMa]

D.5 RPO-Reduce

imports RPO-Tools$^{D.6}$ SN-Booleans$^{D.8}$ Order-Tools$^{H.2}$
exports
context-free syntax
reduce(Tree-m, Tree-m, Order) → BOOL
map(Tree-List-m, Tree-List-m, Order) → BOOL
map-permutations(Permutations-m, Tree-List-m, Order) → BOOL
left2everyright(Tree-m, Tree-List-m, Order) → BOOL
oneleft2right(Tree-List-m, Tree-m, Order) → BOOL
equations

$\text{reduce}(\mathit{Trm},\ \mathit{Trm},\ O) \ = \ \text{true}$ [RRe1]

$$\frac{\text{var}(\mathit{Nm}_1) = \text{true}}{\text{reduce}(\mathit{Nm}_1 - <\ >,\ \mathit{Nm}_2 - < \mathit{Tlm} >,\ O) = \text{eq}(\mathit{Nm}_1,\ \mathit{Nm}_2)}$$ [RRe2]

$$\frac{\text{var}(\mathit{Nm}_2) = \text{true}}{\text{reduce}(\mathit{Nm}_1 - < \mathit{Tlm} >,\ \mathit{Nm}_2 - <\ >,\ O) = \text{in}(\mathit{Nm}_2,\ \mathit{Nm}_1 - < \mathit{Tlm} >)}$$ [RRe3]

$$\frac{\begin{array}{c}\text{var}(\mathit{Nm}) = \text{false},\\ \mathit{Tlm}_1 \neq \mathit{Tlm}_2\end{array}}{\begin{array}{l}\text{reduce}(\mathit{Nm} - < \mathit{Tlm}_1 >,\ \mathit{Nm} - < \mathit{Tlm}_2 >,\ O) = \\ \text{reduce}(\text{mark}(\mathit{Nm} - < \mathit{Tlm}_1 >),\ \mathit{Nm} - < \mathit{Tlm}_2 >,\ O) \\ |\ \text{map-permutations}(\text{permute}(\mathit{Tlm}_1),\ \mathit{Tlm}_2,\ O)\end{array}}$$ [RRe4]

$$\frac{\begin{array}{r}\mathsf{var}(Nm_1) = \mathsf{false},\\ \mathsf{conv}(Nm_1) = Nm_2\end{array}}{\begin{array}{l}\mathsf{reduce}(Nm_1 - < Tlm_1 >, Nm_2 - < Tlm_2 >, O) = \\ \mathsf{map\text{-}permutations}(\mathsf{get\text{-}permute}(\mathsf{number}(Tlm_2), Tlm_1), Tlm_2, O) \\ \mid \mathsf{oneleft2right}(Tlm_1, Nm_2 - < Tlm_2 >, O)\end{array}} \quad \text{[RRe5]}$$

$$\frac{\begin{array}{r}\mathsf{var}(Nm_1) = \mathsf{false},\\ \mathsf{var}(Nm_2) = \mathsf{false},\\ \mathsf{is\text{-}marked}(Nm_1) = \mathsf{false},\\ Nm_1 \neq Nm_2\end{array}}{\begin{array}{l}\mathsf{reduce}(Nm_1 - < Tlm_1 >, Nm_2 - < Tlm_2 >, O) = \\ \mathsf{reduce}(\mathsf{mark}(Nm_1 - < Tlm_1 >), Nm_2 - < Tlm_2 >, O)\end{array}} \quad \text{[RRe6]}$$

$$\frac{\begin{array}{r}\mathsf{var}(Nm_1) = \mathsf{false},\\ \mathsf{var}(Nm_2) = \mathsf{false},\\ \mathsf{is\text{-}marked}(Nm_1) = \mathsf{true},\\ \mathsf{conv}(Nm_1) \neq Nm_2\end{array}}{\begin{array}{l}\mathsf{reduce}(Nm_1 - < Tlm_1 >, Nm_2 - < Tlm_2 >, O) = \\ \mathsf{if}(\mathsf{gt}(Nm_1, Nm_2, O), \mathsf{left2everyright}(Nm_1 - < Tlm_1 >, Tlm_2, O)) \\ \mid \mathsf{oneleft2right}(Tlm_1, Nm_2 - < Tlm_2 >, O)\end{array}} \quad \text{[RRe7]}$$

$\mathsf{map}(, , O) \ = \ \mathsf{true}$ [RMa1]

$\mathsf{map}(Trm_1 \mid Tlm_1, Trm_2 \mid Tlm_2, O) =$ [RMa2]
$\mathsf{reduce}(Trm_1, Trm_2, O)\ \&\ \mathsf{map}(Tlm_1, Tlm_2, O)$

$\mathsf{map}(Tlm_1, Tlm_2, O) \ = \ \mathsf{false}$ **otherwise** [RMa]

$\mathsf{map\text{-}permutations}(, Tlm, O) \ = \ \mathsf{false}$ [RPm1]

$\mathsf{map\text{-}permutations}(Tlm^{+\wedge\wedge}\ Pm^*, Tlm, O) =$ [RPm2]
$\mathsf{map}(Tlm^+, Tlm, O) \mid \mathsf{map\text{-}permutations}(Pm^*, Tlm, O)$

$\mathsf{left2everyright}(Trm, , O) \ = \ \mathsf{true}$ [RLe1]

$\mathsf{left2everyright}(Trm_1, Trm_2 \mid Tlm, O) =$ [RLe2]
$\mathsf{reduce}(Trm_1, Trm_2, O)\ \&\ \mathsf{left2everyright}(Trm_1, Tlm, O)$

$\mathsf{oneleft2right}(, Trm, O) \ = \ \mathsf{false}$ [ROl1]

$\mathsf{oneleft2right}(Trm_1 \mid Tlm, Trm_2, O) =$ [ROl2]
$\mathsf{reduce}(Trm_1, Trm_2, O) \mid \mathsf{oneleft2right}(Tlm, Trm_2, O)$

D.6 RPO-Tools

imports Permutations$^{D.7}$
exports
context-free syntax

embeddable(Tree, Tree-List)	$\to$ BOOL
embed-list(Tree-List, Tree-List)	$\to$ BOOL
embed-permutations(Tree-List, Permutations)	$\to$ BOOL
mark(Tree-m)	$\to$ Tree-m
mark(NAT, Tree-List-m)	$\to$ Tree-List-m
copy(Tree-List-m, Tree-List-m, NAT, BOOL)	$\to$ Permutations-m
copy-marked(Tree-List-m, Tree-List-m, NAT, BOOL)	$\to$ Permutations-m
get-permute(NAT, Tree-List-m)	$\to$ Permutations-m
mark-permute(NAT, Tree-List-m)	$\to$ Permutations-m

hiddens
context-free syntax
if(BOOL, Permutations-m, Permutations-m) $\to$ Permutations-m
equations

$$\text{embeddable}(Tr,\ Tr)\ =\ \text{true} \qquad \text{[REm1]}$$

$$\text{embeddable}(Tr,\ )\ =\ \text{false} \qquad \text{[REm2]}$$

$$\frac{Tl_1 \neq Tl_2}{\begin{array}{l}\text{embeddable}(N - < Tl_1 > ,\ N - < Tl_2 >) = \\ \text{embed-permutations}(Tl_1,\ \text{permute}(Tl_2))\ |\ \text{embeddable}(N - < Tl_1 > ,\ Tl_2)\end{array}} \qquad \text{[REm3]}$$

$$\begin{array}{l}\text{embeddable}(Tr_1,\ Tr_2|\ Tl^+) = \\ \text{embeddable}(Tr_1,\ Tr_2)\ |\ \text{embeddable}(Tr_1,\ Tl^+)\end{array} \qquad \text{[REm4]}$$

$$\begin{array}{l}\text{embeddable}(N_1 - < Tl_1 > ,\ N_2 - < Tl_2 >) = \\ \text{embeddable}(N_1 - < Tl_1 > ,\ Tl_2) \quad \textbf{otherwise}\end{array} \qquad \text{[REm]}$$

$$\text{embed-list}(Tl,\ Tl)\ =\ \text{true} \qquad \text{[REl1]}$$

$$\frac{Tl_1 \neq Tl_2}{\text{embed-list}(Tr|\ Tl_1,\ Tr|\ Tl_2) = \text{embed-list}(Tl_1,\ Tl_2)} \qquad \text{[REl2]}$$

$$\frac{Tr_1 \neq Tr_2}{\begin{array}{l}\text{embed-list}(Tr_1|\ Tl_1,\ Tr_2|\ Tl_2) = \\ \text{embeddable}(Tr_1,\ Tr_2)\ \&\ \text{embed-list}(Tl_1,\ Tl_2)\end{array}} \qquad \text{[REl3]}$$

$$\text{embed-list}(\mathit{Tl}_1,\ \mathit{Tl}_2)\ =\ \text{false}\quad \textbf{otherwise} \qquad \text{[REl]}$$

$$\text{embed-permutations}(\mathit{Tl},\ )\ =\ \text{false} \qquad \text{[REp1]}$$

$$\text{embed-permutations}(\mathit{Tl},\ \mathit{Tl}^{+\wedge\wedge}\ P^*) = \qquad \text{[REp2]}$$
$$\text{embed-list}(\mathit{Tl},\ \mathit{Tl}^+) \mid \text{embed-permutations}(\mathit{Tl},\ P^*)$$

Mark the top node of this tree.

$$\text{mark}((L,\ A,\ \mathit{Tp},\ \text{no-star}) - <\mathit{Tlm}>)\ =\ (L,\ A,\ \mathit{Tp},\ \text{star}) - <\mathit{Tlm}> \qquad \text{[RMa1]}$$

If it is markable, mark the *Nat*-th tree in the list.

$$\frac{\text{markable}(\mathit{Trm}) = \text{true}}{\text{mark}(1,\ \mathit{Trm}|\ \mathit{Tlm}) = \text{mark}(\mathit{Trm})|\ \mathit{Tlm}} \qquad \text{[RMa1]}$$

$$\frac{\mathit{Nat} > 1 = \text{true},\quad \text{mark}(\mathit{Nat} - 1,\ \mathit{Tlm}^+) = \mathit{Tlm}^{+'}}{\text{mark}(\mathit{Nat},\ \mathit{Trm}|\ \mathit{Tlm}^+) = \mathit{Trm}|\ \mathit{Tlm}^{+'}} \qquad \text{[RMa2]}$$

$$\text{mark}(\mathit{Nat},\ \mathit{Tlm})\ =\quad \textbf{otherwise} \qquad \text{[RMa]}$$

Take all possible combinations of *Nat* trees from this list and permute these combinations.

$$\text{get-permute}(0,\ \mathit{Tlm})\ = \qquad \text{[RGp1]}$$

$$\frac{\mathit{Nat} > 0 = \text{true}}{\text{get-permute}(\mathit{Nat},\ \mathit{Tlm}) = \text{copy}(,\ \mathit{Tlm},\ \mathit{Nat},\ \text{false})} \qquad \text{[RGp2]}$$

Put every tree from the source-list zero or more times in the result-list, if it is markable. If it is not markable, it should be copied exactly once. When the result-list contains the required number of elements, all trees left in the source-list have to be markable to marked and copied zero times. If this is the case, the result-list is permuted. When the result-list contains exactly the original source list (this is indicated by *Bool*), one of the trees still has to be marked before permutation can take place.

$$\text{copy}(\mathit{Tlm},\ ,\ 0,\ \mathit{Bool}) = \qquad \text{[RCo1]}$$
$$\text{if}(\mathit{Bool},\ \text{permute}(\mathit{Tlm}),\ \text{mark-permute}(\text{number}(\mathit{Tlm}),\ \mathit{Tlm}))$$

$$\text{copy}(\mathit{Tlm},\ \mathit{Tlm}^+,\ 0,\ \mathit{Bool})\ =\ \text{if}(\text{markable}(\mathit{Tlm}^+),\ \text{permute}(\mathit{Tlm}),\ ) \qquad \text{[RCo2]}$$

$$\frac{\mathit{Nat} > 0 = \text{true}}{\text{copy}(\mathit{Tlm}_1,\ ,\ \mathit{Nat},\ \mathit{Bool}) =} \qquad \text{[RCo3]}$$

$$\frac{\mathit{Nat} > 0 = \text{true},\quad \text{markable}(\mathit{Trm}) = \text{false}}{\text{copy}(\mathit{Tlm}_1,\ \mathit{Trm}|\ \mathit{Tlm}_2,\ \mathit{Nat},\ \mathit{Bool}) = \text{copy}(\mathit{Tlm}_1|\ \mathit{Trm},\ \mathit{Tlm}_2,\ \mathit{Nat} - 1,\ \mathit{Bool})} \qquad \text{[RCo4]}$$

$$\frac{\begin{array}{r}\mathsf{markable}(\mathit{Trm}) = \mathsf{true},\\ \mathsf{copy}(\mathit{Tlm}_1 |\ \mathit{Trm},\ \mathit{Tlm}_2,\ 0,\ \mathit{Bool}) = \mathit{Pm}_1^*,\\ \mathsf{copy}(\mathit{Tlm}_1,\ \mathit{Tlm}_2,\ 1,\ \mathsf{true}) = \mathit{Pm}_2^*\end{array}}{\mathsf{copy}(\mathit{Tlm}_1,\ \mathit{Trm}|\ \mathit{Tlm}_2,\ 1,\ \mathit{Bool}) = \mathit{Pm}_1^{*\wedge\wedge}\ \mathit{Pm}_2^*} \quad \text{[RCo5]}$$

$$\frac{\begin{array}{r}\mathit{Nat} > 1 = \mathsf{true},\\ \mathsf{markable}(\mathit{Trm}) = \mathsf{true},\\ \mathsf{copy\text{-}marked}(\mathit{Tlm}_1 |\ \mathsf{mark}(\mathit{Trm}),\ \mathsf{mark}(\mathit{Trm})|\ \mathit{Tlm}_2,\ \mathit{Nat} - 1,\ \mathsf{true}) = \mathit{Pm}_1^*,\\ \mathsf{copy}(\mathit{Tlm}_1 |\ \mathit{Trm},\ \mathit{Tlm}_2,\ \mathit{Nat} - 1,\ \mathit{Bool}) = \mathit{Pm}_2^*,\\ \mathsf{copy}(\mathit{Tlm}_1,\ \mathit{Tlm}_2,\ \mathit{Nat},\ \mathsf{true}) = \mathit{Pm}_3^*\end{array}}{\mathsf{copy}(\mathit{Tlm}_1,\ \mathit{Trm}|\ \mathit{Tlm}_2,\ \mathit{Nat},\ \mathit{Bool}) = \mathit{Pm}_1^{*\wedge\wedge}\ \mathit{Pm}_2^{*\wedge\wedge}\ \mathit{Pm}_3^*} \quad \text{[RCo6]}$$

Put a marked tree more than once in the result list, until the result-list contains the required number of elements, or the next tree is chosen to be added. When the result-list contains the required number of elements, all trees left in the source-list have to be markable to marked and copied zero times. If this is the case, the result-list is permuted.

$$\mathsf{copy\text{-}marked}(\mathit{Tlm}_1,\ \mathit{Tlm}_2,\ 0,\ \mathit{Bool}) \ = \ \mathsf{copy}(\mathit{Tlm}_1,\ \mathit{Tlm}_2,\ 0,\ \mathit{Bool}) \quad \text{[RCm1]}$$

$$\frac{\mathit{Nat} > 0 = \mathsf{true}}{\mathsf{copy\text{-}marked}(\mathit{Tlm}_1,\ ,\ \mathit{Nat},\ \mathit{Bool}) =} \quad \text{[RCm2]}$$

$$\mathsf{copy\text{-}marked}(\mathit{Tlm}_1,\ \mathit{Trm}|\ \mathit{Tlm}_2,\ 1,\ \mathit{Bool}) \ = \ \mathsf{permute}(\mathit{Tlm}_1 |\ \mathit{Trm}) \quad \text{[RCm3]}$$

$$\frac{\begin{array}{r}\mathit{Nat} > 1 = \mathsf{true},\\ \mathsf{copy\text{-}marked}(\mathit{Tlm}_1 |\ \mathit{Trm},\ \mathit{Trm}|\ \mathit{Tlm}_2,\ \mathit{Nat} - 1,\ \mathit{Bool}) = \mathit{Pm}_1^*,\\ \mathsf{copy}(\mathit{Tlm}_1 |\ \mathit{Trm},\ \mathit{Tlm}_2,\ \mathit{Nat} - 1,\ \mathit{Bool}) = \mathit{Pm}_2^*\end{array}}{\mathsf{copy\text{-}marked}(\mathit{Tlm}_1,\ \mathit{Trm}|\ \mathit{Tlm}_2,\ \mathit{Nat},\ \mathit{Bool}) = \mathit{Pm}_1^{*\wedge\wedge}\ \mathit{Pm}_2^*} \quad \text{[RCm4]}$$

Since all children have been copied without duplication or deletion, the parents mark still has to be pushed down. Try all possibilities.

$$\mathsf{mark\text{-}permute}(1,\ \mathit{Trm}|\ \mathit{Tlm}) \ = \ \mathsf{if}(\mathsf{markable}(\mathit{Trm}),\ \mathsf{permute}(\mathsf{mark}(\mathit{Trm})|\ \mathit{Tlm}),\ ) \quad \text{[RMp1]}$$

$$\frac{\begin{array}{r}\mathit{Nat} > 1 = \mathsf{true},\\ \mathsf{permute}(\mathsf{mark}(\mathit{Nat},\ \mathit{Tlm}^+)) = \mathit{Pm}_1^*,\\ \mathsf{mark\text{-}permute}(\mathit{Nat} - 1,\ \mathit{Tlm}^+) = \mathit{Pm}_2^*\end{array}}{\mathsf{mark\text{-}permute}(\mathit{Nat},\ \mathit{Tlm}^+) = \mathit{Pm}_1^{*\wedge\wedge}\ \mathit{Pm}_2^*} \quad \text{[RMp2]}$$

$$\mathsf{mark\text{-}permute}(\mathit{Nat},\ \mathit{Tlm}) \ = \quad \textbf{otherwise} \quad \text{[RMp]}$$

Conditional function: Uses non-list-construction variables to prevent eager evaluation.

$$\mathsf{if}(\mathsf{true},\ \mathit{Pms}_1,\ \mathit{Pms}_2) \ = \ \mathit{Pms}_1 \quad \text{[RIf1]}$$
$$\mathsf{if}(\mathsf{false},\ \mathit{Pms}_1,\ \mathit{Pms}_2) \ = \ \mathit{Pms}_2 \quad \text{[RIf2]}$$

D.7 Permutations

imports Tree-Tools[F.7] Marked-Tree-Tools[G.3]
exports
sorts Permutations Permutations-m
context-free syntax

{Tree-List "^^"}*	→ Permutations
adding(Tree, Tree-List, Tree-List)	→ Permutations
combinations(Tree, Permutations)	→ Permutations
permute(Tree-List)	→ Permutations
{Tree-List-m "^^"}*	→ Permutations-m
adding(Tree-m, Tree-List-m, Tree-List-m)	→ Permutations-m
combinations(Tree-m, Permutations-m)	→ Permutations-m
permute(Tree-List-m)	→ Permutations-m

variables

P "*"[*0-9′*]*	→ {Tree-List "^^"}*
Ps [*0-9′*]*	→ Permutations
Pm "*"[*0-9′*]*	→ {Tree-List-m "^^"}*
Pms [*0-9′*]*	→ Permutations-m

equations

$$\text{adding}(Tr,\ Tl,\ )\ =\ Tl|\ Tr \qquad \text{[PAd1]}$$

$$\frac{\text{adding}(Tr_1,\ Tl_1|\ Tr_2,\ Tl_2) = P^*}{\text{adding}(Tr_1,\ Tl_1,\ Tr_2|\ Tl_2) = Tl_1|\ Tr_1|\ Tr_2|\ Tl_2\text{\^{}\^{}}\ P^*} \qquad \text{[PAd2]}$$

$$\text{combinations}(Tr,\ )\ =\ \qquad \text{[PCo1]}$$

$$\frac{\begin{array}{c}\text{adding}(Tr,\ ,\ Tl^+) = P^*_1,\\ \text{combinations}(Tr,\ P^*) = P^*_2\end{array}}{\text{combinations}(Tr,\ Tl^{+}\text{\^{}\^{}}\ P^*) = P^*_1\text{\^{}\^{}}\ P^*_2} \qquad \text{[PCo2]}$$

Condition needed for disambiguation, due to overloaded function name permute.

$$\frac{Tl =}{\text{permute}(Tl) =} \qquad \text{[PPe1]}$$

$$\text{permute}(Tr)\ =\ Tr \qquad \text{[PPe2]}$$

$$\text{permute}(Tr|\ Tl^+)\ =\ \text{combinations}(Tr,\ \text{permute}(Tl^+)) \qquad \text{[PPe3]}$$

$$\text{adding}(Trm,\ Tlm,\ )\ =\ Tlm|\ Trm \qquad \text{[MAd1]}$$

$$\frac{\mathsf{adding}(\mathit{Trm}_1,\ \mathit{Tlm}_1|\ \mathit{Trm}_2,\ \mathit{Tlm}_2) = \mathit{Pm}^*}{\mathsf{adding}(\mathit{Trm}_1,\ \mathit{Tlm}_1,\ \mathit{Trm}_2|\ \mathit{Tlm}_2) = \mathit{Tlm}_1|\ \mathit{Trm}_1|\ \mathit{Trm}_2|\ \mathit{Tlm}_2\hat{}\hat{}\ \mathit{Pm}^*} \quad [\mathsf{MAd2}]$$

$$\mathsf{combinations}(\mathit{Trm},\ ) \ = \quad [\mathsf{MCo1}]$$

$$\frac{\begin{array}{c}\mathsf{adding}(\mathit{Trm},\ ,\ \mathit{Tlm}^+) = \mathit{Pm}_1^*,\\ \mathsf{combinations}(\mathit{Trm},\ \mathit{Pm}^*) = \mathit{Pm}_2^*\end{array}}{\mathsf{combinations}(\mathit{Trm},\ \mathit{Tlm}^{+\hat{}\hat{}}\ \mathit{Pm}^*) = \mathit{Pm}_1^{*\hat{}\hat{}}\ \mathit{Pm}_2^*} \quad [\mathsf{MCo2}]$$

Condition needed for disambiguation, due to overloaded function name permute.

$$\frac{\mathit{Tlm} =}{\mathsf{permute}(\mathit{Tlm}) =} \quad [\mathsf{MPe1}]$$

$$\mathsf{permute}(\mathit{Trm}) \ = \ \mathit{Trm} \quad [\mathsf{MPe2}]$$

$$\mathsf{permute}(\mathit{Trm}|\ \mathit{Tlm}^+) \ = \ \mathsf{combinations}(\mathit{Trm},\ \mathsf{permute}(\mathit{Tlm}^+)) \quad [\mathsf{MPe3}]$$

D.8 SN-Booleans

imports Booleans$^{F.6}$
exports
  sorts SN-MSG SN-BOOL
  context-free syntax

“Temporarily” “failing”	→ SN-MSG	
“Failed” “reducing” “eq” “:” Literal	→ SN-MSG	
“All” “orders” “failed”	→ SN-MSG	
“Reduction” “succeeded” “for” “eq” “:” Literal	→ SN-MSG	
“All” “equations” “checked” “using” “order” “:” Order	→ SN-MSG	
“Unconditional” “specification”	→ SN-MSG	
“Not” “implemented” “for” “default-equations”	→ SN-MSG	
“Not” “implemented” “for” “conditional” “specifications”	→ SN-MSG	
BOOL “↔” SN-MSG	→ SN-BOOL	
SN-BOOL “|” SN-BOOL	→ SN-BOOL	{**left**}
SN-BOOL “&” SN-BOOL	→ SN-BOOL	{**left**}
“(” SN-BOOL “)”	→ SN-BOOL	{**bracket**}
if(BOOL, SN-BOOL, SN-BOOL)	→ SN-BOOL	
if(SN-BOOL, SN-BOOL)	→ SN-BOOL	
if(BOOL, BOOL)	→ BOOL	

  variables
    Smsg [*0-9′*]∗ → SN-MSG
    Sbool [*0-9′*]∗ → SN-BOOL
  priorities
    SN-BOOL “|”SN-BOOL → SN-BOOL < SN-BOOL “&”SN-BOOL → SN-BOOL

equations

$$\mathsf{true} \leftrightarrow \mathit{Smsg} \mid \mathit{Sbool} \;=\; \mathsf{true} \leftrightarrow \mathit{Smsg} \qquad [\mathsf{SOr1}]$$
$$\mathsf{false} \leftrightarrow \mathit{Smsg} \mid \mathit{Sbool} \;=\; \mathit{Sbool} \qquad [\mathsf{SOr2}]$$

$$\mathsf{false} \leftrightarrow \mathit{Smsg} \;\&\; \mathit{Sbool} \;=\; \mathsf{false} \leftrightarrow \mathit{Smsg} \qquad [\mathsf{SAnd1}]$$
$$\mathsf{true} \leftrightarrow \mathit{Smsg} \;\&\; \mathit{Sbool} \;=\; \mathit{Sbool} \qquad [\mathsf{SAnd2}]$$

Conditional functions:
If (first argument) then (second argument) else (third argument)

$$\mathsf{if}(\mathsf{true}, \mathit{Sbool}_1, \mathit{Sbool}_2) \;=\; \mathit{Sbool}_1 \qquad [\mathsf{SIf1}]$$
$$\mathsf{if}(\mathsf{false}, \mathit{Sbool}_1, \mathit{Sbool}_2) \;=\; \mathit{Sbool}_2 \qquad [\mathsf{SIf2}]$$

If (first argument) then (second argument) else (first argument)

$$\mathsf{if}(\mathsf{true} \leftrightarrow \mathit{Smsg}, \mathit{Sbool}) \;=\; \mathit{Sbool} \qquad [\mathsf{SIf3}]$$
$$\mathsf{if}(\mathsf{false} \leftrightarrow \mathit{Smsg}, \mathit{Sbool}) \;=\; \mathsf{false} \leftrightarrow \mathit{Smsg} \qquad [\mathsf{SIf4}]$$

If (first argument) then (second argument) else false

$$\mathsf{if}(\mathsf{true}, \mathit{Bool}) \;=\; \mathit{Bool} \qquad [\mathsf{RIf1}]$$
$$\mathsf{if}(\mathsf{false}, \mathit{Bool}) \;=\; \mathsf{false} \qquad [\mathsf{Rif2}]$$

Appendix E

AsFix Terms (data type)

E.1 ATerms

```
imports Literals^E.2
exports
  sorts ATerm
  context-free syntax
    Literal                    → ATerm
    nil                        → ATerm
    ATerm ";" ATerm            → ATerm  {right}
    "[" ATerm ATerm "]"        → ATerm
    ATerm "/" ATerm            → ATerm  {left}
    "(" ATerm ")"              → ATerm  {bracket}
  variables
    T [0-9']*   → ATerm
    Ts [0-9']*  → {ATerm ","}*
    Ts "+"      → {ATerm ","}+
    L [0-9']*   → Literal
  priorities
    ATerm "/" ATerm → ATerm  >  ATerm ";" ATerm → ATerm
```

E.2 Literals

```
exports
  sorts Literal
  lexical syntax
    [␣\t\n]              → LAYOUT
    "%%" ~[\n]* "\n"     → LAYOUT
    "%" ~[\n%]+ "%"      → LAYOUT

    "\" ~□               → EscChar
    "\" [01][0-7][0-7]   → EscChar

    ~[\000-\037"\]       → L-Char
```

EscChar	→	L-Char
"\"" L-Char∗ "\""	→	Literal

E.3 Layout

exports

lexical syntax

"%%" ~[\n]∗[\n]	→	LAYOUT
[␣\t\n]	→	LAYOUT

Appendix F

Rules (data type)

F.1 Rules

imports Tree$^{F.2}$ Booleans$^{F.6}$
exports
  sorts Rule Rules Cond Conds
  context-free syntax

Literal ":" Tree "⟶" Tree "if" Conds	→ Rule
Tree "==" Tree	→ Cond
Tree "<>" Tree	→ Cond
{Cond "&"}∗	→ Conds
Rule∗	→ Rules
default(Literal)	→ BOOL

  variables

"R"[*0-9′*]∗	→ Rule
"R∗"[*0-9′*]∗	→ Rule∗
"C"[*0-9′*]∗	→ Cond
"C∗"[*0-9′*]∗	→ {Cond "&"}∗
"C+"[*0-9′*]∗	→ {Cond "&"}+

hiddens
  variables
    Chars → CHAR∗
equations

default(literal(`"""` `"d"` `"e"` `"f"` `"a"` `"u"` `"l"` `"t"` `"-"` *Chars* `"""`))	= true		[TDe1]
default(*L*)	= false	**otherwise**	[TDe]

F.2 Tree

imports Arity$^{F.4}$ Type$^{F.3}$ Annotations!!
exports
  sorts Node Tree Tree-List

context-free syntax

"(" Literal "," Arity "," Type ")" → Node
Node "− <" Tree-List ">" → Tree
{Tree "|"}∗ → Tree-List

variables

"N"[*0-9′*]∗ → Node
"Tr"[*0-9′*]∗ → Tree
"Tl"[*0-9′*]∗ → {Tree "|"}∗
"Tl+"[*0-9′*]∗ → {Tree "|"}+
L [*0-9′*]∗ → Literal

F.3 Type

imports Literals$^{E.2}$ Iter$^{F.5}$

exports

sorts Type Separator

context-free syntax

Literal → Separator
Literal → Type
Literal Separator Iter → Type

variables

"Tp"[*0-9′*]∗ → Type
"Sp"[*0-9′*]∗ → Separator

F.4 Arity

imports Naturals Iter$^{F.5}$

exports

sorts Arity

context-free syntax

NAT → Arity
Iter → Arity
"Var" → Arity
"MetaVar" → Arity
"Lex" → Arity

variables

"A"[*0-9′*]∗ → Arity

F.5 Iter

exports

sorts Iter

context-free syntax

"∗" → Iter
"+" → Iter

variables

"It"[*0-9′*]∗ → Iter

F.6 Booleans

imports Layout$^{E.3}$
exports
sorts BOOL
context-free syntax

true	$\rightarrow$ BOOL	
false	$\rightarrow$ BOOL	
BOOL "|" BOOL	$\rightarrow$ BOOL	{**left**}
BOOL "&" BOOL	$\rightarrow$ BOOL	{**left**}
not(BOOL)	$\rightarrow$ BOOL	
"(" BOOL ")"	$\rightarrow$ BOOL	{**bracket**}

variables
Bool [*0-9'*]$*$ $\rightarrow$ BOOL
priorities
"|" $<$ BOOL "&" BOOL $\rightarrow$ BOOL

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]

F.7 Tree-Tools

imports Tree$^{F.2}$ Booleans$^{F.6}$
exports
context-free syntax

eq(Node, Node)	$\rightarrow$ BOOL
eq(Tree, Tree)	$\rightarrow$ BOOL
eq-all(Tree, Tree-List)	$\rightarrow$ BOOL
type(Node)	$\rightarrow$ Type
func(Node)	$\rightarrow$ BOOL
list(Node)	$\rightarrow$ BOOL
var(Node)	$\rightarrow$ BOOL
metavar(Node)	$\rightarrow$ BOOL

$\mathsf{lex(Node)} \rightarrow \mathsf{BOOL}$
$\mathsf{in(Node, Tree\text{-}List)} \rightarrow \mathsf{BOOL}$
$\mathsf{in\text{-}all(Node, Tree\text{-}List)} \rightarrow \mathsf{BOOL}$

equations

$\mathsf{eq}(N, N) = \mathsf{true}$ [TEn1]
$\mathsf{eq}(N_1, N_2) = \mathsf{false}$ **otherwise** [TEn]

$\mathsf{eq}(Tr, Tr) = \mathsf{true}$ [TEt1]
$\mathsf{eq}(Tr_1, Tr_2) = \mathsf{false}$ **otherwise** [TEqt]

Check whether a tree is equal to all trees in a tree-list.

$\mathsf{eq\text{-}all}(Tr,) = \mathsf{false}$ [TEa1]
$\mathsf{eq\text{-}all}(Tr_1, Tr_2) = \mathsf{eq}(Tr_1, Tr_2)$ [TEa2]
$\mathsf{eq\text{-}all}(Tr_1, Tr_2 | \; Tl^+) = \mathsf{eq}(Tr_1, Tr_2) \;\&\; \mathsf{eq\text{-}all}(Tr_1, Tl^+)$ [TEa3]

$\mathsf{type}((L, A, Tp)) = Tp$ [TTy1]

$\mathsf{func}((L, Nat, Tp)) = \mathsf{true}$ [TFu1]
$\mathsf{func}(N) = \mathsf{false}$ **otherwise** [TFun]

$\mathsf{list}((L, *, Tp)) = \mathsf{true}$ [TLi1]
$\mathsf{list}((L, +, Tp)) = \mathsf{true}$ [TLi2]
$\mathsf{list}(N) = \mathsf{false}$ **otherwise** [TLi]

$\mathsf{lex}((L, \mathsf{Lex}, Tp)) = \mathsf{true}$ [TLe1]
$\mathsf{lex}(N) = \mathsf{false}$ **otherwise** [TLe]

$\mathsf{var}((L, \mathsf{Var}, Tp)) = \mathsf{true}$ [TVa1]
$\mathsf{var}(N) = \mathsf{false}$ **otherwise** [TVa]

$\mathsf{metavar}((L, \mathsf{MetaVar}, Tp)) = \mathsf{true}$ [TMe1]
$\mathsf{metavar}(N) = \mathsf{false}$ **otherwise** [TMe]

Check whether this node appears in the tree-list.

$\mathsf{in}(N,) = \mathsf{false}$ [TIn1]
$\mathsf{in}(N_1, N_2 - < Tl >) = \mathsf{eq}(N_1, N_2) \mid \mathsf{in}(N_1, Tl)$ [TIn2]
$\mathsf{in}(N, Tr | \; Tl^+) = \mathsf{in}(N, Tr) \mid \mathsf{in}(N, Tl^+)$ [TIn3]

Check whether this node appears in all trees in the tree-list.

$\mathsf{in\text{-}all}(N,) = \mathsf{false}$ [TIa1]
$\mathsf{in\text{-}all}(N_1, N_2 - < Tl >) = \mathsf{in}(N_1, N_2 - < Tl >)$ [TIa2]
$\mathsf{in\text{-}all}(N, Tr | \; Tl^+) = \mathsf{in}(N, Tr) \;\&\; \mathsf{in\text{-}all}(N, Tl^+)$ [TIa3]

F.8 Make-Rules

imports Make-Tree$^{F.9}$ Rules$^{F.1}$ Tree-Tools$^{F.7}$
exports
context-free syntax
asfix2tree(ATerm) $\rightarrow$ Rules
conditions(ATerm) $\rightarrow$ Conds
equations

Translate AsFix-CondEquations in an AsFix-Module to Rules.
For details about the AsFix format, see [Kli93].
Everything that does not match the right AsFix format is skipped. This eliminates e.g. metavariables of the form <CondEquation>.

The last list-argument of a "Module" application contains the equations that we need.

$$\text{asfix2tree}([\texttt{"Module"}\ L_1;\ L_2;\ T_1;\ T_2;\ T_3;\ T_4]) = \text{asfix2tree}(T_4) \quad \text{[MAt1]}$$

$$\text{asfix2tree}([\texttt{"PrefixEquations"}\ T]) = \text{asfix2tree}(T) \quad \text{[MAt2]}$$

Take from a "CondEquation" application the label, left-hand-side, right-hand-side and conditions.

$$\text{asfix2tree}([\texttt{"CondEquation"}\ L_1;\ L_2;\ T_1;\ T_2;\ [\texttt{"Conditions"}\ T_3]]) = L_2 : \text{make-tree}(T_1) \longrightarrow \text{make-tree}(T_2)\ \text{if conditions}(T_3) \quad \text{[MAt3]}$$

A list of equations yields a lists of rules.

$$\frac{\text{asfix2tree}(T_1) = R_1^*,\quad \text{asfix2tree}(T_2) = R_2^*}{\text{asfix2tree}(T_1;\ T_2) = R_1^*\ R_2^*} \quad \text{[MAt4]}$$

$$\text{asfix2tree}(T) = \quad \textbf{otherwise} \quad \text{[MAt]}$$

A positive condition.

$$\text{conditions}([\texttt{"Condition"}\ \texttt{"Eq"};\ T_1;\ T_2]) = \text{make-tree}(T_1) == \text{make-tree}(T_2) \quad \text{[MCo1]}$$

A negative condition.

$$\text{conditions}([\texttt{"Condition"}\ \texttt{"Neq"};\ T_1;\ T_2]) = \text{make-tree}(T_1) <> \text{make-tree}(T_2) \quad \text{[MCo2]}$$

A list of conditions.

$$\frac{\text{conditions}(T_1) = C_1^*,\quad \text{conditions}(T_2) = C_2^*}{\text{conditions}(T_1;\ T_2) = C_1^* \&\ C_2^*} \quad \text{[MCo3]}$$

$$\text{conditions}(T) = \quad \textbf{otherwise} \quad \text{[MCo]}$$

F.9 Make-Tree

imports ATerms[E.1] Tree[F.2] Booleans[F.6]
exports
  context-free syntax
    make-tree(ATerm) → Tree
    get-name(ATerm) → Literal
    get-arity(ATerm) → NAT
    get-iter(Literal) → Iter
    get-type(ATerm) → Type
    get-children(ATerm) → Tree-List
hiddens
  context-free syntax
    concat(Literal, Literal) → Literal
  variables
    Chars [*0-9′*]∗ → CHAR∗
equations

make-tree(["Var" *L*; *T*]) = (*L*, Var, get-type(*T*)) − < > [MMt1]
make-tree(["MetaVar" *L*; *T*]) = (*L*, MetaVar, get-type(*T*)) − < > [MMt2]
make-tree(["Lex" *L*; *T*]) = (*L*, Lex, get-type(*T*)) − < > [MMt3]

make-tree([["CfFunction" T_1; T_2; T_3] T_4]) = [MMt4]
(get-name(T_1), get-arity(T_1), get-type(T_2)) − < get-children(T_4) >

make-tree([["Iter" ["Sort" L_1]; L_2; L_3] T_2]) = [MMt5]
(L_2, get-iter(L_3), L_1 L_2 get-iter(L_3)) − < get-children(T_2) >

make-tree([["Iter" ["Sort" L_1]; ["QLit" L_2]; L_3] T_2]) = [MMt6]
(L_2, get-iter(L_3), L_1 L_2 get-iter(L_3)) − < get-children(T_2) >

Construct the name of a function out of the CFElems.

get-name(*L*) = *L* [MGn1]
get-name(*L*; *T*) = concat(*L*, get-name(*T*)) [MGn2]
get-name(["Sort" *L*]) = "" [MGn3]
get-name(["Sort" *L*]; *T*) = get-name(*T*) [MGn4]
get-name(["QLit" *L*]) = *L* [MGn5]
get-name(["QLit" *L*]; *T*) = concat(*L*, get-name(*T*)) [MGn6]
get-name(["Iter" *T*]) = "" [MGn7]
get-name(["Iter" T_1]; T_2) = get-name(T_2) [MGn8]
get-name(["CfElems" *T*]) = get-name(*T*) [MGn9]
get-name(*T*) = "error-get-name" **otherwise** [MGn]

Count the number of arguments in the CfElems.

$$\text{get-arity}([\texttt{"CfElems"}\ T]) = \text{get-arity}(T) \qquad \text{[MGa1]}$$
$$\text{get-arity}([\texttt{"Sort"}\ L]) = 1 \qquad \text{[MGa2]}$$
$$\text{get-arity}([\texttt{"Sort"}\ L];\ T) = 1 + \text{get-arity}(T) \qquad \text{[MGa3]}$$
$$\text{get-arity}([\texttt{"QLit"}\ L];\ T) = \text{get-arity}(T) \qquad \text{[MGa4]}$$
$$\text{get-arity}([\texttt{"Iter"}\ T]) = 1 \qquad \text{[MGa5]}$$
$$\text{get-arity}([\texttt{"Iter"}\ T_1];\ T_2) = 1 + \text{get-arity}(T_2) \qquad \text{[MGa6]}$$
$$\text{get-arity}(L;\ T) = \text{get-arity}(T) \qquad \text{[MGa7]}$$
$$\text{get-arity}(T) = 0 \quad \textbf{otherwise} \qquad \text{[MGa]}$$

Get the iteration-sort.

$$\text{get-iter}(\texttt{"*"}) = * \qquad \text{[MGi1]}$$
$$\text{get-iter}(\texttt{"+"}) = + \qquad \text{[MGi2]}$$

Find the output-type.

$$\text{get-type}([\texttt{"Sort"}\ L]) = L \qquad \text{[MGt1]}$$
$$\text{get-type}([\texttt{"Iter"}\ [\texttt{"Sort"}\ L_1];\ L_2;\ \texttt{"*"}]) = L_1\ L_2 * \qquad \text{[MGt2]}$$
$$\text{get-type}([\texttt{"Iter"}\ [\texttt{"Sort"}\ L_1];\ L_2;\ \texttt{"+"}]) = L_1\ L_2 + \qquad \text{[MGt3]}$$
$$\text{get-type}([\texttt{"Iter"}\ [\texttt{"Sort"}\ L_1];\ [\texttt{"QLit"}\ L_2];\ \texttt{"*"}]) = L_1\ L_2 * \qquad \text{[MGt4]}$$
$$\text{get-type}([\texttt{"Iter"}\ [\texttt{"Sort"}\ L_1];\ [\texttt{"QLit"}\ L_2];\ \texttt{"+"}]) = L_1\ L_2 + \qquad \text{[MGt5]}$$
$$\text{get-type}(T) = \texttt{"error-get-type"} \quad \textbf{otherwise} \qquad \text{[MGt]}$$

Build tree-lists from arguments.

$$\text{get-children}(\text{nil}) = \qquad \text{[MGc1]}$$

$$\frac{\text{get-children}(T_1) = Tl_1,\quad \text{get-children}(T_2) = Tl_2}{\text{get-children}(T_1;\ T_2) = Tl_1 \mid Tl_2} \qquad \text{[MGc2]}$$

$$\text{get-children}(T) = \text{make-tree}(T) \quad \textbf{otherwise} \qquad \text{[MGc]}$$

concat: Constructs a name, given a number of literals.

$$\text{concat}(\text{literal}(\texttt{"""}\ \mathit{Chars}_1\ \texttt{"""}),\ \texttt{"err-get-name"}) = \texttt{"err-get-name"} \qquad \text{[MCo1]}$$

$$\text{concat}(\text{literal}(\texttt{"""}\ \mathit{Chars}_1\ \texttt{"""}),\ \text{literal}(\texttt{"""}\ \mathit{Chars}_2\ \texttt{"""})) = \text{literal}(\texttt{"""}\ \mathit{Chars}_1\ \mathit{Chars}_2\ \texttt{"""}) \quad \textbf{otherwise} \qquad \text{[MCo]}$$

Appendix G

Marked Tree (data type)

G.1 Marked-Tree

imports ATerms$^{E.1}$ Naturals Arity$^{F.4}$ Type$^{F.3}$ Mark$^{G.2}$
exports
  sorts Node-m Tree-m Tree-List-m
  context-free syntax
    "(" Literal "," Arity "," Type "," Mark ")" → Node-m
    Node-m "− <" {Tree-m "|"}∗ ">" → Tree-m
    {Tree-m "|"}∗ → Tree-List-m
  variables
    "Nm"[*0-9′*]∗ → Node-m
    "Trm"[*0-9′*]∗ → Tree-m
    "Tlm"[*0-9′*]∗ → {Tree-m "|"}∗
    "Tlm+"[*0-9′*]∗ → {Tree-m "|"}+

G.2 Mark

exports
  sorts Mark
  context-free syntax
    "star" → Mark
    "no-star" → Mark
  variables
    "M"[*0-9′*]∗ → Mark

G.3 Marked-Tree-Tools

imports Marked-Tree$^{G.1}$ Tree$^{F.2}$
exports
  context-free syntax
    tree2mtree(Tree) → Tree-m
    treelist2mtreelist(Tree-List) → Tree-List-m
    is-marked(Node-m) → BOOL

eq(Node-m, Node-m)	$\rightarrow$	BOOL
eq(Tree-m, Tree-m)	$\rightarrow$	BOOL
eq-all(Tree-m, Tree-List-m)	$\rightarrow$	BOOL
func(Node-m)	$\rightarrow$	BOOL
list(Node-m)	$\rightarrow$	BOOL
var(Node-m)	$\rightarrow$	BOOL
lex(Node-m)	$\rightarrow$	BOOL
in(Node-m, Tree-List-m)	$\rightarrow$	BOOL
in-all(Node-m, Tree-List-m)	$\rightarrow$	BOOL
number(Tree-List-m)	$\rightarrow$	NAT
conv(Node-m)	$\rightarrow$	Node-m
markable(Tree-List-m)	$\rightarrow$	BOOL

equations

Add space for a mark to every node in the tree

$$\frac{\text{treelist2mtreelist}(Tl) = Tlm}{\text{tree2mtree}((L, A, Tp) - < Tl >) = (L, A, Tp, \text{no-star}) - < Tlm >} \quad \text{[MTm1]}$$

$$\text{treelist2mtreelist}() = \quad \text{[MTm2]}$$

$$\frac{\text{treelist2mtreelist}(Tl) = Tlm}{\text{treelist2mtreelist}(Tr \mid Tl) = \text{tree2mtree}(Tr) \mid Tlm} \quad \text{[MTm3]}$$

$$\text{is-marked}((L, A, Tp, \text{star})) = \text{true} \quad \text{[MIm1]}$$
$$\text{is-marked}((L, A, Tp, \text{no-star})) = \text{false} \quad \text{[MIm2]}$$

$$\text{eq}(Nm, Nm) = \text{true} \quad \text{[Meq1]}$$
$$\text{eq}(Nm_1, Nm_2) = \text{false} \quad \textbf{otherwise} \quad \text{[Meq]}$$

$$\text{eq}(Trm, Trm) = \text{true} \quad \text{[Meq2]}$$
$$\text{eq}(Trm_1, Trm_2) = \text{false} \quad \textbf{otherwise} \quad \text{[Meq]}$$

$$\text{eq-all}(Trm,) = \text{false} \quad \text{[MEa1]}$$
$$\text{eq-all}(Trm_1, Trm_2) = \text{eq}(Trm_1, Trm_2) \quad \text{[MEa2]}$$
$$\text{eq-all}(Trm_1, Trm_2 \mid Tlm^+) = \text{eq}(Trm_1, Trm_2)\ \&\ \text{eq-all}(Trm_1, Tlm^+) \quad \text{[MEa3]}$$

$$\text{func}((L, Nat, Tp, M)) = \text{true} \quad \text{[MFu1]}$$
$$\text{func}(Nm) = \text{false} \quad \textbf{otherwise} \quad \text{[MFu]}$$

$$\text{list}((L, *, Tp, M)) = \text{true} \quad \text{[MLi1]}$$
$$\text{list}((L, +, Tp, M)) = \text{true} \quad \text{[MLi2]}$$
$$\text{list}(Nm) = \text{false} \quad \textbf{otherwise} \quad \text{[MLi]}$$

$$\begin{array}{lll} \mathsf{lex}((L, \mathsf{Lex}, Tp, M)) & = \mathsf{true} & \text{[MLe1]} \\ \mathsf{lex}(Nm) & = \mathsf{false} \quad \textbf{otherwise} & \text{[MLe]} \end{array}$$

$$\begin{array}{lll} \mathsf{var}((L, \mathsf{Var}, Tp, M)) & = \mathsf{true} & \text{[MVa1]} \\ \mathsf{var}((L, \mathsf{MetaVar}, Tp, M)) & = \mathsf{true} & \text{[MVa2]} \\ \mathsf{var}(Nm) & = \mathsf{false} \quad \textbf{otherwise} & \text{[MVa]} \end{array}$$

Check whether this node appears in the tree-list.

$$\begin{array}{lll} \mathsf{in}(Nm,) & = \mathsf{false} & \text{[MIn1]} \\ \mathsf{in}(Nm_1, Nm_2 - < Tlm >) & = \mathsf{eq}(Nm_1, Nm_2) \mid \mathsf{in}(Nm_1, Tlm) & \text{[MIn2]} \\ \mathsf{in}(Nm, Trm|\ Tlm^+) & = \mathsf{in}(Nm, Trm) \mid \mathsf{in}(Nm, Tlm^+) & \text{[MIn3]} \end{array}$$

Check whether this node appears in all trees in the tree-list.

$$\begin{array}{lll} \mathsf{in\text{-}all}(Nm,) & = \mathsf{false} & \text{[MIa1]} \\ \mathsf{in\text{-}all}(Nm_1, Nm_2 - < Tlm >) & = \mathsf{in}(Nm_1, Nm_2 - < Tlm >) & \text{[MIa2]} \\ \mathsf{in\text{-}all}(Nm, Trm|\ Tlm^+) & = \mathsf{in}(Nm, Trm)\ \&\ \mathsf{in\text{-}all}(Nm, Tlm^+) & \text{[MIa3]} \end{array}$$

Count the number of trees in a list.

$$\begin{array}{lll} \mathsf{number}() & = 0 & \text{[MNu1]} \\ \mathsf{number}(Trm|\ Tlm) & = 1 + \mathsf{number}(Tlm) & \text{[MNu2]} \end{array}$$

Mark an unmarked node, unmark a marked node.

$$\begin{array}{lll} \mathsf{conv}((L, A, Tp, \mathsf{star})) & = (L, A, Tp, \mathsf{no\text{-}star}) & \text{[MCo1]} \\ \mathsf{conv}((L, A, Tp, \mathsf{no\text{-}star})) & = (L, A, Tp, \mathsf{star}) & \text{[MCo2]} \end{array}$$

Only non-variable nodes are markable.

$$\begin{array}{lll} \mathsf{markable}() & = \mathsf{false} & \text{[MMa1]} \\ \mathsf{markable}(Nm - < Tlm_1 >) & = \mathsf{not}(\mathsf{var}(Nm)) & \text{[MMa2]} \\ \mathsf{markable}(Trm|\ Tlm^+) & = \mathsf{markable}(Trm)\ \&\ \mathsf{markable}(Tlm^+) & \text{[MMa3]} \end{array}$$

Appendix H

Orders (data type)

H.1 Orders

imports Tree$^{F.2}$
exports
  sorts Orders Order Pair
  context-free syntax
    Node ">" Node → Pair
    "{" {Pair ","}∗ "}" → Order
    {Order "#"}∗ → Orders
  variables
    "P"[0-9′]∗ → Pair
    "P∗"[0-9′]∗ → {Pair ","}∗
    "O"[0-9′]∗ → Order
    "O∗"[0-9′]∗ → {Order "#"}∗
    "O+"[0-9′]∗ → {Order "#"}+
    "Os"[0-9′]∗ → Orders
equations

$$\{P_1^*, N_1 > N_2, P_2^*, N_1 > N_2, P_3^*\} = \{P_1^*, N_1 > N_2, P_2^*, P_3^*\} \quad \text{[Or1]}$$

¡¡order.eq¿¿=

$$O_1^* \# \ O \# \ O_2^* \# \ O \# \ O_3^* = O_1^* \# \ O \# \ O_2^* \# \ O_3^* \quad \text{[Or2]}$$

@

H.2 Order-Tools

imports Orders$^{H.1}$ Marked-Tree$^{G.1}$ Booleans$^{F.6}$
exports
  context-free syntax
    eq(Orders, Orders) → BOOL
    eq(Pair, Pair) → BOOL
    if(BOOL, Orders, Orders) → Orders

$$\begin{array}{ll}
\mathsf{in(Pair,\ Order)} & \to \mathsf{BOOL} \\
\mathsf{gt(Node\text{-}m,\ Node\text{-}m,\ Order)} & \to \mathsf{BOOL} \\
\mathsf{add(Pair,\ Order)} & \to \mathsf{Order} \\
\mathsf{trans(Order)} & \to \mathsf{Order} \\
\mathsf{trans\text{-}close(Order)} & \to \mathsf{Order} \\
\mathsf{trans\text{-}close\text{-}more(Orders)} & \to \mathsf{Orders}
\end{array}$$

equations

$$\mathsf{eq}(O^*,\ O^*) \ = \ \mathsf{true} \qquad [\mathsf{Eq1}]$$

$$\mathsf{eq}(O_1^*,\ O_2^*) \ = \ \mathsf{false} \quad \textbf{otherwise} \qquad [\mathsf{eq}]$$

$$\mathsf{eq}(P,\ P) \ = \ \mathsf{true} \qquad [\mathsf{OEq1}]$$

$$\mathsf{eq}(P_1,\ P_2) \ = \ \mathsf{false} \quad \textbf{otherwise} \qquad [\mathsf{Eq}]$$

Since we are only interested in reduction of the chosen orders, we use variables of the sort Orders instead of the sort {Order"#"}∗ to prevent eager evaluation.

$$\mathsf{if}(\mathsf{true},\ Os_1,\ Os_2) \ = \ Os_1 \qquad [\mathsf{OIf1}]$$

$$\mathsf{if}(\mathsf{false},\ Os_1,\ Os_2) \ = \ Os_2 \qquad [\mathsf{OIf2}]$$

$$\mathsf{in}(P,\ \{P_1^*,\ P,\ P_2^*\}) \ = \ \mathsf{true} \qquad [\mathsf{OIn1}]$$

$$\mathsf{in}(P,\ O) \ = \ \mathsf{false} \quad \textbf{otherwise} \qquad [\mathsf{OIn}]$$

Check whether the first node is greater than the second in this order.

$$\mathsf{gt}((L_1,\ A_1,\ Tp_1,\ M_1),\ (L_2,\ A_2,\ Tp_2,\ M_2),\ O) \ = \ \mathsf{in}((L_1,\ A_1,\ Tp_1) > (L_2,\ A_2,\ Tp_2),\ O) \qquad [\mathsf{OGt1}]$$

Add a pair to an order if it is not already there.

$$\frac{\mathsf{in}(P,\ \{P^*\}) = \mathsf{false}}{\mathsf{add}(P,\ \{P^*\}) = \{P,\ P^*\}} \qquad [\mathsf{OAd1}]$$

$$\mathsf{add}(P,\ O) \ = \ O \quad \textbf{otherwise} \qquad [\mathsf{OAd}]$$

Find all transitive relations, given an order.

$$\mathsf{trans}(\{\}) \ = \ \{\} \qquad [\mathsf{OTr1}]$$

$$\mathsf{trans}(\{P\}) \ = \ \{\} \qquad [\mathsf{OTr2}]$$

$$\frac{\begin{array}{c} N_1 \neq N_3, \\ \mathsf{trans}(\{N_1 > N_2,\ P_1^*\}) = \{P_2^*\}, \\ \mathsf{trans}(\{N_2 > N_3,\ P_1^*\}) = \{P_3^*\}, \\ \mathsf{trans}(\{N_1 > N_3,\ P_1^*\}) = \{P_4^*\} \end{array}}{\mathsf{trans}(\{N_1 > N_2,\ N_2 > N_3,\ P_1^*\}) = \{N_1 > N_3,\ P_2^*,\ P_3^*,\ P_4^*\}} \qquad [\mathsf{OTr3}]$$

$$\frac{\begin{array}{c} N_2 \neq N_3, \\ \mathsf{trans}(\{N_1 > N_2,\, P_1^*\}) = \{P_2^*\}, \\ \mathsf{trans}(\{N_3 > N_1,\, P_1^*\}) = \{P_3^*\}, \\ \mathsf{trans}(\{N_3 > N_2,\, P_1^*\}) = \{P_4^*\} \end{array}}{\mathsf{trans}(\{N_1 > N_2,\, N_3 > N_1,\, P_1^*\}) = \{N_3 > N_2,\, P_2^*,\, P_3^*,\, P_4^*\}} \quad \text{[OTr4]}$$

$$\frac{\begin{array}{c} N_1 \neq N_4, \quad N_2 \neq N_3, \\ \mathsf{trans}(\{N_1 > N_2,\, P_1^*\}) = \{P_2^*\}, \\ \mathsf{trans}(\{N_3 > N_4,\, P_1^*\}) = \{P_3^*\} \end{array}}{\mathsf{trans}(\{N_1 > N_2,\, N_3 > N_4,\, P_1^*\}) = \{P_2^*,\, P_3^*\}} \quad \text{[OTr5]}$$

Compute the transitive closure of an order, by adding all transitive relations.

$$\frac{\mathsf{trans}(\{P_1^*\}) = \{P_2^*\}}{\mathsf{trans\text{-}close}(\{P_1^*\}) = \{P_1^*,\, P_2^*\}} \quad \text{[OTc1]}$$

Bibliography

[BKM89] J.A. Bergstra, J.W. Klop, and A. Middeldorp. *Termherschrijfsystemen*. Kluwer Bedrijfswetenschappen, 1989. In Dutch.

[DOS88] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 538–549. Springer-Verlag, 1988.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as SDFmanual.ps.Z.

[Kli93] P. Klint, editor. *The ASF+SDF Meta-environment User's Guide,* version 26 February. Centrum voor Wiskunde en Informatica (CWI), 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as SysManual.ps.Z.

[Kli94] P. Klint. Writing meta-level specifications in ASF+SDF. unpublished report, Centrum voor Wiskunde en Informatica (CWI), 1994.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[Knu73] D. E. Knuth. *The art of computer programming*, volume 3: Sorting and searching. Addison-Wesley, 1973.

[KZ89] D. Kapur and H. Zhang. *RRL: Rewrite Rule Laboratory - User's manual*, 1989. Revised version. Available by *ftp* from cs.uiowa.edu:/pub/rrl as rrlmanual.tar.Z.