Combinatory Algebraic Specification & Compilation of List Matching

Eelco Visser

Doctoraalscriptie Informatica
Universiteit van Amsterdam
Afstudeerdocent: prof. dr. P. Klint
Begeleiders: dr. H.R. Walters & drs. J.F.Th. Kamperman
Examendatum: 18 juni 1993

Colofon

Typeset in ASF+SDF+$\mathcal{AMS}$-LaTeX by the author
Figures programmed in TPIC
Coverdesign by R.M. McColl

Acknowledgements

This thesis is the result of a graduation research project conducted at the Programming Research Group of the University of Amsterdam from October 1992 until May 1993. The starting point of the project was the design and implementation of a component for the ASF+SDF compiler, which is being developed at the CWI in Amsterdam by Pum Walters and Jasper Kamperman.

Paul Klint introduced me to the project and supported my excursion into Combinatory Algebraic Specification. Pum Walters en Jasper Kamperman guided me with persistent, but constructive, criticism on all aspects of my work.

Arie van Deursen joined me in the work on the elimination of conditional rewrite rules from term rewrite systems. The brainstorm sessions yielded an idea for the transformation, but, due to other priorities, the planned paper never came beyond draft status. Arie also handed me the idea how to handle comment in literate specifications. Wilco Koorn answered numerous questions about the meta-environment, SEAL and 'Centaur programming'. Jan Heering read a very early version of Part I and suggested using something else than 'higher-order' to name this style of algebraic specification. Khalil Sima'an and Gert Vos read drafts of Part I.

The GIPE group at CWI and the Programming Research Group of the University of Amsterdam create a stimulating environment for research. The (almost) weekly technical meetings bring new ideas and lively discussions.

Fien McColl designed the cover for this thesis, encouraged me (not to exceed deadlines too much), and, above all, shared three lovely years with me.

Thank you all

Eelco Visser
Amsterdam
May 1993

Contents

List of Figures

Chapter 1

Introduction

This thesis contains an algebraic specification in ASF+SDF of a component of a compiler for term rewrite systems (TRSs). The component transforms TRSs with rewrite rules over terms with associative lists to equivalent TRSs without rules over lists. The specification of the component is written in an extension of ASF+SDF called *Combinatory Algebraic Specification*.

In this chapter we give a short introduction to ASF+SDF and the ASF+SDF compiler project. Then we formulate the goals of this thesis and introduce the solutions for them. The chapter is concluded with a discussion of literate programming in ASF+SDF.

1.1 ASF+SDF

1.1.1 The Formalism

ASF+SDF (the combination of the formalisms ASF and SDF) is an algebraic specification formalism for the specification of syntax and semantics of (programming) languages.

ASF (*Algebraic Specification Formalism*) [BHK89] is a formalism for the definition of many sorted algebra's. A specification consists of a *signature* and a set of (conditional) *equations*. The signature specifies a set of *sorts*, a set of *functions* over the sorts, and a set of *variables* over the sorts. The equations identify (open) terms over the signature.

Since specifications of non-trivial problems can become quite large, specifications can be divided into *modules*. The signature of a module is divided in an *exports* and a *hiddens* part. When a module is *imported* into another module, only the exported part of the signature is visible in that module.

The model of a specification is a many sorted *algebra* (or family of algebras) such that, there is a mapping from each sort in the signature to a set in the algebra, and a mapping from each function in the signature to a function over these sets. The algebra should satisfy the interpretation under that maping of the universal closure of each equation in the specification.

Usually the intended model of a specification is its *initial algebra*. The elements of the sorts of the initial algebra are equivalence classes of terms. Two terms are equivalent if they are provably equal in conditional equational logic with the equations of the specification as axioms.

SDF (*Syntax Definition Formalism*) [HHKR89, HK89] is a formalism for the definition of context-free languages. A definition consists of a set of *non-terminals*, a set of *lexical* and *context-free* grammar rules, and a set of *priority rules*. The model of an SDF definition is a mapping of non-terminals to sets of strings. In the combination with ASF, non-terminals are interpreted as sorts, grammar-rules as functions, and sentences over the grammar as terms.

1.1.2 The Implementation

The ASF+SDF meta-environment [Kli90] is an interactive environment for the development, generation and testing of programming environments.

The SDF part of a specification is used to generate *lazy, incremental parsers* [Rek92] and *syntax-directed editors* [Koo92] for the languages defined.

The ASF part is interpreted as a *term rewrite system* (TRS), by orienting the equations from left to right, which can be used to *reduce* (i.e. compute the *normal form* of) a term [Hen91, Wal91, Wie91]. The reduction algorithm of the interpreter uses a leftmost-innermost reduction strategy.

1.2 The ASF+SDF Compiler

The meta-environment can be used to test programming environments generated from a specification. It can not yet be used for the automatic generation of stand-alone programming environments. Furthermore the reduction of terms by the interpreter is too slow for big computations. The first step towards the generation of stand-alone environments is the generation of a reduction machine for the rewrite rules in a particular specification.

The ASF+SDF compiler project has developed a compiler that translates algebraic specifications to imperative programs. The development of the compiler is described in [WK93]. In this section we give a short outline of the structure of the compiler. In the next section we then introduce the ideas for restructuring the compiler that were the starting-point for the research presented in this thesis.

The compiler, called `ASFtoC` (Figure 1.1), translates ASF+SDF specifications to `C` programs. The `C` program resulting from a compilation can be compiled with a `C`-compiler to an executable program, that can be used to reduce terms over the signature of the specification.

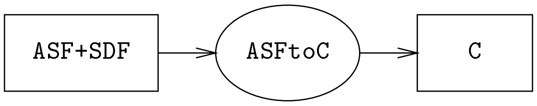


Figure 1.1: The ASF+SDF compiler.

1.2.1 Intermediate Languages

The compiler uses the intermediate languages RN1 and ARM.

RN1 (*Rule Notation 1*) is a formalism with the expressive power of ASF (with respect to term rewriting) but has no sorts, modules, or free syntax. All functions are in prefix notation with disambiguated names. No overloading of functions is allowed.

The only non-standard feature in RN1 is the associative list function; that is, a special function with a variable number of arguments (of the same sort). The focus of the research presented here is on the removal of this special function from RN1 specifications. A detailed discussion will be given in Section 8.1.

RN1 was designed with the property that RN1 specifications can be read as RN1 terms. Therefore RN1 specifications can be written that manipulate RN1 specifications. A property that ASF+SDF lacks because of the free syntax of terms.

A definition of (syntax and semantics of) RN2, a version of RN1 better suited for human reading, will be given in Chapters 9, 10 and 11.

ARM (*Abstract Reduction Machine*) is an abstract machine dedicated to term rewriting. It has two stacks, the control stack and the argument stack, and a heap. The control stack contains the term to be processed, the argument stack keeps track of normal forms of subterms of the original term. The trees representing the normal forms are kept in the heap. The machine has the property that only terms in normal form are represented as trees. Non-normalized terms are kept on the control stack.

The machine is driven by an ARM program, which represents the rewrite system.

Since we will not be concerned with ARM in this thesis, we will not further discuss it. A detailed discussion of ARM can be found in [KW93].

1.2.2 Components of the Compiler

We discuss the four phases (excluding compilation of the resulting `C` program) in the compilation of specifications; the front-end translates a specification to a TRS, three transformations normalize the TRS, which is then compiled to machine code for the Abstract Reduction Machine [KW93]. Finally this machine code is compiled into `C` code.

Front End An ASF+SDF specification is translated to an RN1 specification of an untyped term rewrite system by `ASFtoRN1` (Figure 1.2).

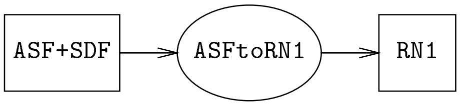


Figure 1.2: Front-end of the ASF+SDF compiler.

Preprocessing Before the actual compilation, three preliminary operations are performed on the RN1 specification (Figure 1.3).

`Tc` ('typecheck') checks the RN1 code for violations of several statical rules. This is not needed when the code is generated by `ASFtoRN1`, because the original ASF specification is checked by the meta-environment. If typechecking does not succeed, an error message is given.

`GenIO` adds functions for input and output. The resulting executable can read an RN1 term from standard input as a list of characters. This list is parsed by the function *$i*, yielding the input term. The term is normalized and then converted to a list of characters by the function *$o*.

`PrePro` does some simple transformations on RN1 specifications. Positive conditions are divided in proper positive conditions (not introducing new variables) and 'new-var' conditions (introducing new variables). Specifications are made left-linear, possibly adding conditions to rewrite rules. Matching terms (left-hand sides of equations and left-hand sides of new-var conditions) containing (nested) occurrences of the list function are transformed, so that the list function only appears as the outermost function symbol of new-var conditions.

Left-linearity, matching terms and nested lists will be discussed in more detail in Part II.

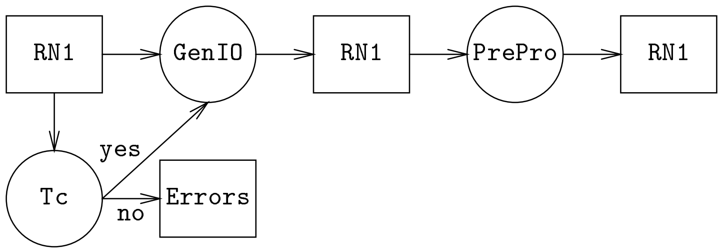


Figure 1.3: Preprocessing of `RN1`.

Compilation An RN1 specification is translated to ARM machine code by `RN1toARM` (Figure 1.4).

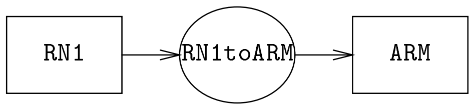


Figure 1.4: Compilation of `RN1` to `ARM`.

Back End Finally, after some optimizations, the ARM code is translated to a C program (Figure 1.5).

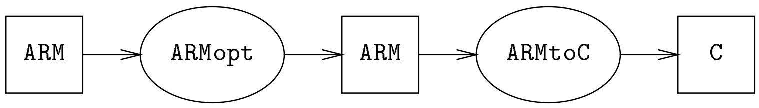


Figure 1.5: Back-End of the ASF+SDF compiler.

The resulting executable can be used from the meta-environment to reduce terms. The term is first translated to an RN1 term, then fed to the executable, and the normal form is translated back to the original syntax.

1.3 Restructuring the Compiler

The following observations were made during the design and implementation of the compiler as outlined above.

- Most complications in the implementation of ARM are caused by lists. Memory management is complicated since a list occupies a non-fixed amount of space. In contrast, prefix functions with a fixed arity always occupy the same amount of space. A lot of bookkeeping must be done to keep track of the last match for a list pattern.
- ARM would be simpler if it did not have to handle conditions, since the machine must do a lot of bookkeeping to know where to continue when the validation of the conditions of a rule fails.
- Transformations on RN1 have proven to be successful tools in writing the compiler.

These observations led to a redesign of the compiler with more emphasis on transformations of RN1 specifications.

In the new set-up of the compiler (Figure 1.6) the compilation phase is replaced by a series of transformations (`Trans`) that transform RN1 specifications to specifications of rewrite systems with only unconditional rules over constants, and unary and binary functions without associative lists. The resulting micro-RN1 specification is then compiled to a spartan version of ARM, micro-ARM, which does not support the primitives for list matching and conditional rewriting.

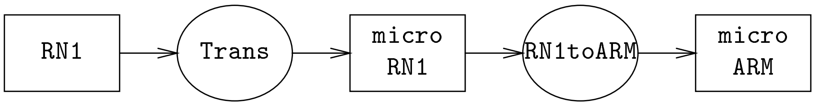


Figure 1.6: New Structure of Compilation Phase.

The resulting ARM code is much simpler and more uniform. This makes optimizations to the code more effective and easier since less cases have to be dealt with. Furthermore the trees that are built in the micro-ARM machine have a fixed arity, which simplifies memory management.

Goals This led to the formulation of the following goals for the research project described in this thesis.

1. Try to capture common operations used in transformations and describe them in a generic way.

2. Develop transformations on RN1 specifications that eliminate list expressions and conditional rewrite rules and that preserve the efficiency (with respect to both space and time) of the original specifications.

3. Specify these transformations in ASF+SDF and integrate the compiled specifications as new compiler components in the compiler.

Results The first goal led to a method of algebraic specification in which higher-order functions can be used; Combinatory Algebraic Specification. The latter goals led to a specification, using the combinatory tools, of the transformation of RN2 specifications with associative lists to equivalent RN2 specifications without lists.

Combinatory Algebraic Specification Part I of this thesis provides a tools package based on the language of *combinatory terms*; a simple *applicative* language with built-in syntax for *lists*, *tuples* and *infix operators*. Constructs from other languages can be embedded in this language by declaring them as *constants*. Higher-order functions or *combinators* can be added to the language. The semantics of combinators is defined with ASF+SDF equations.

The package defines an extensive library of combinators, including general combinators for the manipulation of functions, and many general purpose list and tree operations. Furthermore, the language is extended with λ-abstraction and Miranda's *list comprehension* construct.

Combinatory Algebraic Specification is further introduced in Chapter 2.

Transformations In Part II the TRS specification formalism RN2 is defined. The syntax of RN2 is defined as the extension of Basic-RN2 with three extensions, conditions, default rules and associative lists. The language is embedded in the combinatory setting and basic combinators for construction and analysis of RN2 constructs are defined. With this basis we define the syntax and (static and dynamic) semantics of RN2. Then three, semantically conservative, transformations on RN2 are specified that, together, eliminate associative lists from specifications.

A further introduction to RN2 and transformations, as well as an overview of Part II is given in Chapter 8.

1.4 Literate Programming in ASF+SDF+LaTeX

We conclude this chapter with some remarks about the layout of this thesis.

The idea of programs that are their own documentation is known as *literate programming*. This term stems from Knuth who introduced it in [Knu84]. In that article a new attitude towards programming is proposed. Computer programs should not only be instructions for computers but also explanations to human beings of what the computer is supposed to do.

To this end Knuth designed the language `WEB`. Programs in `WEB` can be translated to Pascal programs for compilation (`TANGLE`) and to TeX programs for typesetting (`WEAVE`). The programmer is supposed to develop the program while explaining it to the reader. To support this, a `WEB` program does not have to follow the rigid rules of Pascal; the programmer can defer the definition of a part of the program (e.g. a difficult algorithm) until a later stage in the explanation.

Since the subject of this thesis is a large algebraic specification (about 50 modules), a literate programming approach is used to explain it. In our case this means that specification and explanation are merged.

The source of this thesis is a legal ASF+SDF specification that can be executed by the meta-environment as a term rewrite system. The modules of the specification are translated to LaTeX programs for typesetting by TeX by the ToLaTeX program written by Paul Klint. The ToLaTeX program was adapted, such that the comment, included in modules, is typeset as normal text. The translated modules are glued together by means of a LaTeX file that groups the modules in chapters and adds a table of contents, introductions, conclusions, a bibliography etc..

We give a short description of the features of the ToLaTeX program. ('$\mapsto$' is a meta-symbol of this explanation, showing how a specification fragment is typeset.)

Literals beginning with a '\' are recognized as TeX commands. For instance,

$$\texttt{\textbackslash lambda\textbackslash Var.\textbackslash Var = \textbackslash I} \quad \mapsto \quad \lambda\mathcal{V}.\mathcal{V} = \mathbf{I}$$

ASCII Symbols like `->` have special meaning to the program. The program has a table mapping ASCII combinations to TeX commands. For instance, '`->`' is mapped to '$\rightarrow$' and '`|[`' is mapped to '$[\![$'. So we have

$$\texttt{\textbackslash trs |[\textbackslash Term]|} \quad \mapsto \quad \mathcal{T}[\![t]\!]$$

Fonts ASF+SDF keywords, literals, tags and variables are distinguished by setting them in different fonts. If a variable ends in a number, the number is set as subscript to the variable.

Equations are displayed. Equations may be centered or set aligned left. Equations may be labeled with the tag from the specification, a numeric tag or not at all.

Terms are broken by the programmer, the indentation is done by the ToLaTeX program.

Comment is usually written after the comment signs %%. The ToLaTeX program includes these comments in the LaTeX file after stripping the comment signs. For instance,

`%% The next equation shows`
`%% how we finally make the`
`%% specification inconsistent.`
`%%%`
	$\mapsto$	The next equation shows how we finally make the specification inconsistent.

`[T-F] \True = \False`		$\top = \bot$

If the comment includes items that may be mistaken for ASF+SDF constructs (such as equations) they should be marked so. In this thesis we mark example equations which are not part of the specification by a ★.

Module Headings are formed by a section number and the name of the module. The keyword 'module' is left out. The name of the module in the heading may deviate somewhat from the actual name (as it appears in imports sections) because module names may not contain spaces. For instance, module `Combinatory-Logic` has the heading

4.2 Combinatory Logic

Import Section The names of imported modules are annotated with the section in which the imported module is defined.

`imports Combinatory-Logic` $\mapsto$ **Imports** Combinatory-Logic$^{(4.2)}$

When the annotation is $^{(??)}$ the imported module is not included in this thesis.

Part I

Combinatory Algebraic Specification

Chapter 2

Introduction to Part I

2.1 Reusability of Specifications

The development of reusable software components is a well known practice in software engineering to reduce the effort of software development. The formalism ASF+SDF, which we use in this thesis, has a module construct. Specifications are sets of modules. A module can be reused by importing it in a new module. However, the reusability is constrained by the strict typing mechanism of ASF+SDF. For instance, if we have a module specifying tables that map identifiers to integers, we can not reuse this module to define tables that map ISBN numbers to book titles, if the sorts ISBN number and identifier, respectively, book title and integer are not the same.

Parameterized modules are the common answer to these problems in algebraic specification. The formalism ASF has a parameter mechanism. With this mechanism sorts and functions can be made parameters of a module. Importing modules fill in the parameters, thereby obtaining a module tailored to the needs of the module.

In the implementation of ASF+SDF the parameter mechanism is not (yet) included. The only modules that are used as library modules by many specifications are the Boolean and integer modules. Other common datatypes have to be spelled out for each version. For instance, a typical function over a list sort has the equations

$$\begin{aligned} f([]) &= [] \\ f([X, X*]) &= [Y, Y*] \Leftarrow Y = g(Y), [Y*] = f([X*]) \end{aligned}$$

All elements of the list are visited and changed by the function g. For a function f' that applies a function g' to the elements of a list, we have to write the same equations with f replaced by f' and g by g'.

2.2 A Solution

The function f above can be specified as

$$f([X*]) = map(g, [X*])$$

where map is a specified as

$$\begin{aligned} map(F, []) &= [] \\ map(F, [X, X*]) &= [Y, Y*] \Leftarrow Y = F(Y), [Y*] = map(F, [X*]) \end{aligned}$$

Map is a higher-order function that has a function (F) as its argument. Now a recursion over lists has been expressed in a generic way. Every function that needs this recursion can be defined in terms of *map*.

The basic idea of this approach is that functions should be considered as objects that can be combined to form new functions without reference to their arguments.

A problem with this approach is that *map* has the signature

$$map\text{“(”}\alpha \rightarrow \beta\text{“,”}\alpha^*\text{“)”} \rightarrow \beta^*$$

where α and β are type variables. However, this can not be expressed in the first-order formalism ASF+SDF.

An Observation Untyped Combinatory Logic [CF58, CHS72] is a first-order term rewrite system. The system **CL** has two constant Terms **S** and **K** and one function, *application*,

$$app(\text{Term}, \text{Term}) \rightarrow \text{Term}$$

The axioms or reduction rules for **S** and **K** are

$$\begin{aligned} app(app(app(\mathbf{S}, X), Y), Z) &= app(app(X, Z), app(Y, Z)) \\ app(app(\mathbf{K}, X), Y) &= X \end{aligned}$$

With **S** functions can be combined into new functions and with **K** objects can be thrown away. This system has universal computational power. All recursive functions can be expressed by terms built from **S** and **K**. For instance, composition of unary functions is expressed by

$$app(app(\mathbf{S}, app(\mathbf{K}, \mathbf{S})), \mathbf{K})$$

Combinatory Logic forms the basis of the implentation of some functional programming languages [Pey87].

A Conclusion Since the function *app* is a first-order function, Combinatory Logic can be expressed in ASF+SDF. With the syntactic facilities of SDF we can even define the function more elegantly as

$$\text{Term Term} \rightarrow \text{Term } \{\textbf{left}\}$$

which defines application as the left-associative juxtaposition of two Terms.

The Result In Chapter 3 we define a simple applicative language, inspired by Miranda [BW88], in which constants and combinators can be combined by application to form complex terms. The language is extended with built in syntax for infix operators, lists, tuples and λ-abstraction and Miranda's list comprehensions.

To aid the reader in understanding the combinators, a system of type annotations of combinators is used. For instance, the combinator *map* can be declared as

$$map \rightarrow \text{Combinator } \{\textbf{type:}\ (\alpha \rightarrow \beta) \rightarrow \alpha^* \rightarrow \beta^*\}$$

with the reduction rules

$$\begin{aligned} &map\ F\ [] &&= [] \\ &map\ F\ [X, X*] &&= [Y, Y*] \Leftarrow Y = F\ Y, [Y*] = map\ F\ [X*] \end{aligned}$$

As we see in this example, combinators are not defined in terms of **S** and **K**, but by new reduction rules.

The specification of the syntax is extended with an extensive library of general purpose combinators.

Chapter 4 defines the basic combinators of Combinatory Logic, several general purpose combinators on functions, λ-abstraction, and the abstract data types Boolean, Natural, List, Tuple, Character and String. Chapter 5 defines an extensive library of general purpose combinators on lists. Chapter 6 defines the abstract data type Tree and several functions on trees.

Module Tools, shown below, is the collection of all utilities developed in this part and may be imported in a specification as a tools library. The import graph for this module is shown in Figure 2.1.

Finally, in Chapter 7 we will evaluate this method of algebraic specification, discuss some related work and present ideas for future work.

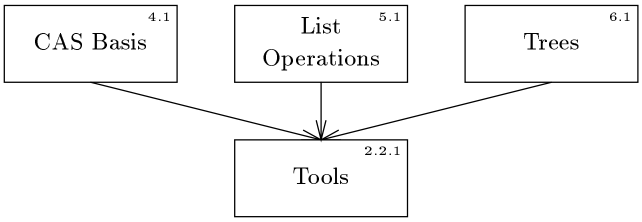


Figure 2.1: Import graph for Chapter 2. Tools

2.2.1 Tools

Imports CAS-basis$^{(4.1)}$ List-Operations$^{(5.1)}$ Trees$^{(6.1)}$

Basic combinators, list operations and tree operations are combined in a tools package.

Chapter 3

Syntax

3.1 Combinatory Terms

Imports Layout$^{(??)}$

First, we lay the syntactic basis for the untyped functional programming language we will further explore in the following modules.

Terms are built from constants and combinators with application, []-lists and $\langle\rangle$-tuples. Constants are terms over first-order sorts defined outside the scope of this specification that are imported into the language of combinatory terms.

Exports
  Sorts Constant Combinator Term
  Context-free Functions
    Constant $\rightarrow$ Term
    Combinator $\rightarrow$ Term

    Term Term $\rightarrow$ Term {**left**}

    "[" {Term "," }* "]" $\rightarrow$ Term
    "$\langle$" {Term "," }* "$\rangle$" $\rightarrow$ Term

    "(" Term ")" $\rightarrow$ Term {**bracket**}
  Variables
    "δ" [*0-9′*]* $\rightarrow$ Constant
    "$\Box$" [*0-9′*]* $\rightarrow$ Combinator

Application associates to the left; $(\Box\ \delta_1\ \delta_2)$ should be read as $((\Box\ \delta_1)\ \delta_2)$.

Special syntax is used for infix operators like $+$ and $-$. Two varieties are defined: left associative and right associative infix operators. Furthermore, postfix operators may be defined through sort Postfix-Operator.

Exports
  Sorts Infix-Operator Right-Infix-Operator Postfix-Operator
  Context-free Functions
    Term Infix-Operator Term $\rightarrow$ Term {**left**}
    Term Right-Infix-Operator Term $\rightarrow$ Term {**right**}
    Term Postfix-Operator $\rightarrow$ Term
  Variables
    "$\odot$" [*0-9′*]* $\rightarrow$ Infix-Operator
    "$\circledcirc$" [*0-9′*]* $\rightarrow$ Right-Infix-Operator
    "$\boxdot$" [*0-9′*]* $\rightarrow$ Postfix-Operator
  Priorities

Term Term → Term > Term Postfix-Operator → Term >
{ Term Infix-Operator Term → Term,
Term Right-Infix-Operator Term → Term }

Operators can be used as combinators by putting brackets around them. For instance, if + is an Infix-Operator, then (+) is the associated combinator. Partial applications of operators are expressed by the sectioning syntax; (+1) is the function that adds one to its argument, and (1−) is the function that adds one to the inverse of its argument. The semantics of this syntax is given in the equations section below.

Exports

Context-free Functions

"(" Infix-Operator ")"	→ Combinator
"(" Term Infix-Operator ")"	→ Term
"(" Infix-Operator Term ")"	→ Term
"(" Right-Infix-Operator ")"	→ Combinator
"(" Term Right-Infix-Operator ")"	→ Term
"(" Right-Infix-Operator Term ")"	→ Term
"(" Postfix-Operator ")"	→ Combinator

Hiddens

Variables

$[XY]$ $[0\text{-}9']*$ → Term

Equations

Semantics of combinatorized and sectioned infix operators.

$$
\begin{array}{lll}
(1)\ (\odot)\ X\ Y = X \odot Y & (2)\ (X \odot)\ Y = X \odot Y & (3)\ (\odot\ X)\ Y = Y \odot X \\
(4)\ (\circledcirc)\ X\ Y = X \circledcirc Y & (5)\ (X \circledcirc)\ Y = X \circledcirc Y & (6)\ (\circledcirc\ X)\ Y = Y \circledcirc X \\
(7)\ (\boxdot)\ X \quad = X \boxdot & &
\end{array}
$$

3.2 Higher-Order Algebraic Model Desired

This approach gives a very unsatisfactory initial algebra. All terms over the language we have defined are members of the sort Term. We intend a more sophisticated model in which constants of the same type are in a different sort than functions over those constants; i.e. $\mathbb{N}$ is an other sort than $\mathbb{N} \to \mathbb{N}$ which is again different from $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$. Or, 1 should not be in the same sort as (+1) which should be different from (>). But (+1) and (> 1) should both be in the class of unary functions (of type $\alpha \to \beta$).

However, typed combinatory logic ($\mathbf{CL}^{\tau}$ [Bet91]) can not be defined syntactically in ASF+SDF.

To overcome this problem we have chosen to work with untyped combinatory terms to define the (rewrite) semantics of functions, and annotation of the declaration of these functions with higher-order types to express the intended algebraic sort.

The syntax of higher-order types, used in annotations, is defined in the next section.

Remark Higher-Order types could help in expressing semantics of programming languages. Now a Pascal program is a syntactic object that is manipulated by functions.

We could, however, see a Pascal program as a function on its environment (the operating system, for instance). Thus, Pascal programs have type Env $\rightarrow$ Env, where Env is the type of the run-time environment.

A transformation on a program is then a higher-order function of type (Env $\rightarrow$ Env) $\rightarrow$ (Env $\rightarrow$ Env).

3.3 Higher-Order Types

Imports Naturals$^{(??)}$

Here, we define the syntax of higher-order types, that we will use to annotate the declarations of combinators in the rest of this specification. Note that these annotations are only very suggestive comments, but have no formal status.

Basic types are type names and parameterized types.

Exports
  Sorts HO-Type
  Lexical Syntax
    [A-Z] [A-Za-z]∗ $\rightarrow$ HO-Type
  Context-free Functions
    HO-Type "(" {HO-Type "," }∗ ")" $\rightarrow$ HO-Type
    "(" HO-Type ")" $\rightarrow$ HO-Type {**bracket**}
  Variables
    "α" *[0-9′]*∗ $\rightarrow$ HO-Type
    "β" *[0-9′]*∗ $\rightarrow$ HO-Type
    "γ" *[0-9′]*∗ $\rightarrow$ HO-Type
    "$\vec{\alpha}$" *[0-9′]*∗ $\rightarrow$ {HO-Type "," }∗
    "$\vec{\beta}$" *[0-9′]*∗ $\rightarrow$ {HO-Type "," }∗
    "$\vec{\gamma}$" *[0-9′]*∗ $\rightarrow$ {HO-Type "," }∗

The following type constructors build functional types, products, unions, star-lists and plus-lists, respectively.

Exports
  Context-free Functions
    HO-Type "$\rightarrow$" HO-Type $\rightarrow$ HO-Type {**right**}
    HO-Type "$\times$" HO-Type $\rightarrow$ HO-Type {**assoc**}
    HO-Type "$\cup$" HO-Type $\rightarrow$ HO-Type {**assoc**}
    HO-Type "∗" $\rightarrow$ HO-Type
    HO-Type "+" $\rightarrow$ HO-Type

Type-Sets are sequences of types. Type sets can be concatenated with $+\!\!+$ and the n-th element can be projected from a set by postfixing n to the set.

Exports
  Sorts HO-Type-Set
  Context-free Functions
    "{" {HO-Type "," }∗ "}" $\rightarrow$ HO-Type-Set
    HO-Type-Set "$+\!\!+$" HO-Type-Set $\rightarrow$ HO-Type-Set {**left**}
    HO-Type-Set NAT $\rightarrow$ HO-Type
    "(" HO-Type-Set ")" $\rightarrow$ HO-Type-Set {**bracket**}
  Variables
    "Γ" *[0-9′]*∗ $\rightarrow$ HO-Type-Set
    "n" *[0-9′]*∗ $\rightarrow$ NAT

The prefix operators $\prod$, $\bigcup$ and $\uparrow$ translate type sets to products, unions and functional types, respectively. The translation is given in the equations section below. The special type ε (empty type) is needed to express the translation of these operations for empty type sets.

Exports

Context-free Functions

"ε" $\rightarrow$ HO-Type

"$\prod$" HO-Type-Set $\rightarrow$ HO-Type
"$\bigcup$" HO-Type-Set $\rightarrow$ HO-Type
"$\uparrow$" HO-Type-Set $\rightarrow$ HO-Type

To get the n-th type of a product the following index function is defined.

Exports

Context-free Functions

HO-Type NAT $\rightarrow$ HO-Type

Hiddens

Variables

[c] [+] $\rightarrow$ CHAR+

Equations

The empty type ε is a neutral element for type constructors.

$$\begin{array}{llll} \varepsilon \rightarrow \alpha = \alpha & \varepsilon \times \alpha = \alpha & \varepsilon \cup \alpha = \alpha & \varepsilon * = \varepsilon \\ \alpha \rightarrow \varepsilon = \alpha & \alpha \times \varepsilon = \alpha & \alpha \cup \varepsilon = \alpha & \varepsilon + = \varepsilon \end{array}$$

Concatenation of HO-Type-Sets.

$$\{\vec{\alpha}\} +\!\!+ \{\vec{\beta}\} = \{\vec{\alpha}, \vec{\beta}\}$$

The prefix operators $\prod$, $\bigcup$ and $\uparrow$ construct type expressions from type sets.

$$\begin{array}{ll} \prod \{\} = \varepsilon & \prod \{\alpha, \vec{\alpha}\} = \alpha \times \prod \{\vec{\alpha}\} \\ \bigcup \{\} = \varepsilon & \bigcup \{\alpha, \vec{\alpha}\} = \alpha \cup \bigcup \{\vec{\alpha}\} \\ \uparrow \{\} = \varepsilon & \uparrow \{\alpha, \vec{\alpha}\} = \alpha \rightarrow \uparrow \{\vec{\alpha}\} \end{array}$$

For instance, the type $\alpha \rightarrow \beta \rightarrow \gamma$ can also be written as $\uparrow \{\alpha, \beta, \gamma\}$.

Indexing type sets is defined as:

$$\{\alpha, \vec{\alpha}\}\ 0 = \alpha \qquad \{\vec{\alpha}\}\ n = \{\vec{\alpha}\}\ n - 1 \Leftarrow n > 0 = \mathit{true}$$

The n-th element of a product type.

$$\alpha \times \beta\ 0 = \alpha \qquad \alpha \times \beta\ n = \beta\ n - 1 \Leftarrow n > 0 = \mathit{true}$$

For other type the indexing function is undefined.

3.4 Combinatory Terms with Type Annotations

Imports Layout$^{(??)}$

We repeat the syntax of combinatory terms with type annotations. In a special *type attribute* the type deduction for a syntax rule is defined. The types of constructs on the left-hand side are inherited by the notation *type*(S), where S is a sort. If more than one sort with the same name appears we subscript the sorts counting from left to right. Note that this notation is not formally defined in module Higher-Order Types.

In all following modules we will annotate the declarations of constants and combinators similarly.

Exports
Sorts Constant Combinator Term
Context-free Functions

$$\begin{array}{lll}
\text{Constant} & \to \text{Term} & \{\textbf{type: } \text{type(Constant)}\} \\
\text{Combinator} & \to \text{Term} & \{\textbf{type: } \text{type(Combinator)}\} \\
\text{Term Term} & \to \text{Term} & \{\textbf{left}, \textbf{type: } \left.\begin{array}{l}\text{type}(\text{Term}_1) = \alpha \to \beta, \\ \text{type}(\text{Term}_2) = \alpha\end{array}\right\} \Rightarrow \beta\} \\
\text{“[” \{Term “,” \}* “]”} & \to \text{Term} & \{\textbf{type: } \text{type}(\text{Term}^*) = \Gamma \Rightarrow (\bigcup \Gamma)^*\} \\
\text{“⟨” \{Term “,” \}* “⟩”} & \to \text{Term} & \{\textbf{type: } \text{type}(\text{Term}^*) = \Gamma \Rightarrow \prod \Gamma\}
\end{array}$$

The definition for Right-Infix-Operator has been left out since it is the same as for Infix-Operator.

Exports
Sorts Infix-Operator Postfix-Operator
Context-free Functions

$$\begin{array}{lll}
\text{Term Infix-Operator Term} & \to \text{Term} & \{\textbf{left}, \textbf{type: } \left.\begin{array}{l}\text{type(Infix-Operator)} = \alpha \times \beta \to \gamma, \\ \text{type}(\text{Term}_1) = \alpha, \\ \text{type}(\text{Term}_2) = \beta\end{array}\right\} \Rightarrow \gamma\} \\
\text{Term Postfix-Operator} & \to \text{Term} & \{\textbf{type: } \left.\begin{array}{l}\text{type(Postfix-Operator)} = \alpha \to \beta, \\ \text{type(Term)} = \alpha\end{array}\right\} \Rightarrow \beta\}
\end{array}$$

Combinatorizing an infix operator changes its type from $\alpha \times \beta \to \gamma$ to $\alpha \to \beta \to \gamma$.

Exports
Context-free Functions

$$\begin{array}{lll}
\text{“(” Infix-Operator “)”} & \to \text{Combinator} & \{\textbf{type: } \text{type(Infix-Operator)} = \alpha \times \beta \to \gamma\} \\
& & \Rightarrow \alpha \to \beta \to \gamma \\
\text{“(” Postfix-Operator “)”} & \to \text{Combinator} & \{\textbf{type: } \text{type(Postfix-Operator)} = \alpha \to \beta\} \\
& & \Rightarrow \alpha \to \beta
\end{array}$$

Left and right sectioning of infix operators yields a functional type. Note the type of the right-sectioned operator.

Exports
Context-free Functions

$$\begin{array}{lll}
\text{“(” Term Infix-Operator “)”} & \to \text{Term} & \{\textbf{type: } \left.\begin{array}{l}\text{type(Infix-Operator)} = \alpha \times \beta \to \gamma, \\ \text{type(Term)} = \alpha\end{array}\right\} \Rightarrow \beta \to \gamma\} \\
\text{“(” Infix-Operator Term “)”} & \to \text{Term} & \{\textbf{type: } \left.\begin{array}{l}\text{type(Infix-Operator)} = \alpha \times \beta \to \gamma, \\ \text{type(Term)} = \beta\end{array}\right\} \Rightarrow \alpha \to \gamma\}
\end{array}$$

3.5 Extensional versus Intensional Equality

We now have attached types to our combinators but there remains a problem in the models we get. The functions (+1) and (1+) are different objects in the initial algebra of this specification, although they represent both the function that adds one to its argument. The objects are extensionally equal.

In general, two functions are extensionally equal if for each possible argument they yield the same value. That is, F and G are equal if

$$\forall X(FX = GX)$$

The following deduction rule states this for functions in general.

$$\forall F, G(\forall X(FX = GX) \Rightarrow F = G)$$

It is clear that we can not add such a rule to this specification. To decide whether two functions are equal in this sense, an enumeration of all terms in a sort has to be done. Since most sorts contain infinitely many terms the computation will not terminate.

However, in special cases we know that two objects represent the same function. We can then write an equation that reduces one to the other. This can be an advantage if one of the objects is a more efficient representation of the function than the other.

For most functions we will not do this. This means that the model of the specification is intensional. We have to be careful with inequality of functional objects. Two syntactically equal representations of a function are clearly equal. But, if two objects are not syntactically equal, they may still represent the same function.

Chapter 4

Basic Combinators

We start with the definition of a series of basic combinators, among which the standard combinators from Combinatory Logic, the basic datatypes Booleans, naturals, characters and strings, the datatypes lists and tuples, and the definition of λ abstraction.

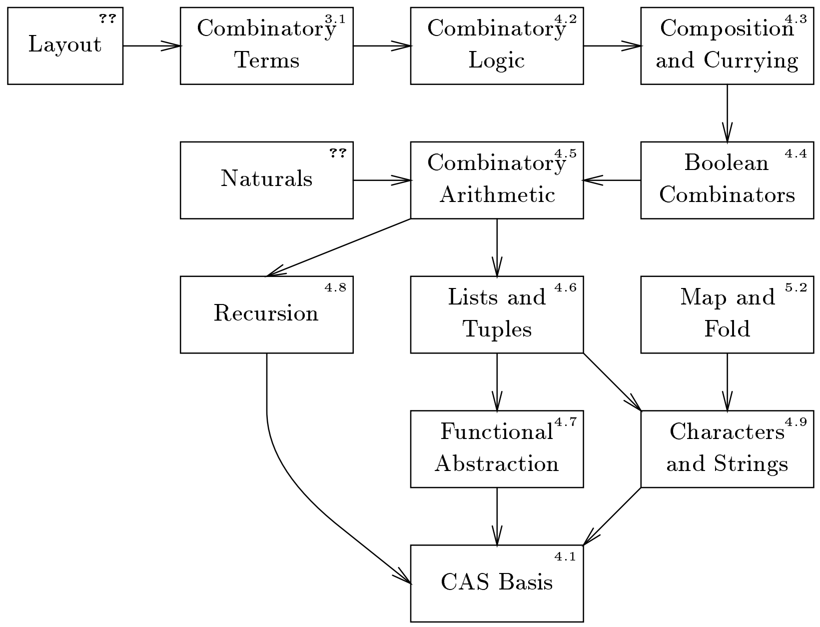


Figure 4.1: Import graph for Chapter 4 Basic Combinators

4.1 CAS Basis

Imports Recursion$^{(4.8)}$ Functional-Abstraction$^{(4.7)}$ Characters-and-Strings$^{(4.9)}$
This module merely collects all modules in this chapter.

4.2 Combinatory Logic

Imports Combinatory-Terms$^{(3.1)}$

Here, we define the combinators from the first sections of Chapter 5 (Intuitive Theory of Combinators) of [CF58].

Exports

Context-free Functions

"**S**" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$}
"**K**" $\rightarrow$ Combinator {**type:** $\alpha \rightarrow \beta \rightarrow \alpha$}
"**I**" $\rightarrow$ Combinator {**type:** $\alpha \rightarrow \alpha$}
"**B**" $\rightarrow$ Combinator {**type:** $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$}
"**C**" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \alpha \rightarrow \gamma$}
"**W**" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$}
"**Φ**" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\delta \rightarrow \alpha) \rightarrow (\delta \rightarrow \beta) \rightarrow \delta \rightarrow \gamma$}
"**Ψ**" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \gamma \rightarrow \beta$}

Hiddens

Variables

[*FGHMNXYZ*] $\rightarrow$ Term

Equations

Standard Combinators Combinator **I** is the identity function.

$$\mathbf{I}\ X = X \tag{I}$$

Combinator **K**, the *elementary cancellator*, turns an object X into a unary function that always yields X.

$$\mathbf{K}\ X\ Y = X \tag{K}$$

Combinators **S** turns a binary function F and a unary function G into a new unary function.

$$\mathbf{S}\ F\ G\ X = F\ X\ (G\ X) \tag{S}$$

Combinator **B** defines the composition of two unary functions F and G.

$$\mathbf{B}\ F\ G\ X = F\ (G\ X) \tag{B}$$

Combinator **C** swaps the arguments of a binary function F.

$$\mathbf{C}\ F\ X\ Y = F\ Y\ X \tag{C}$$

For instance, if $(-)$ defines subtraction, then $\mathbf{C}(-)$ subtracts its first argument from its last.

W turns a binary function F into a unary function that duplicates its argument and passes them to F.

$$\mathbf{W}\ F\ X = F\ X\ X \tag{W}$$

If $(\cdot)$ defines multiplication, then $\mathbf{W}(\cdot)$ is the function that squares its argument.

The combinator Φ is analogous to **B**. Where **B** applies a function G before applying F, Φ applies functions G and H to its argument before applying F to the results.

$$\Phi\ F\ G\ H\ X = F\ (G\ X)\ (H\ X) \tag{Phi}$$

For instance, if $(\rightarrow)$ is Boolean implication, then $\Phi(\rightarrow)$ is the implication of unary predicates. Thus we have

$$\Phi(\rightarrow)\ P_1\ P_2\ X \twoheadrightarrow (P_1\ X \rightarrow P_2\ X)$$

Finally, combinator Ψ uses a function G to preprocess the arguments of a binary function F.

$$\Psi\ F\ G\ X\ Y = F\ (G\ X)\ (G\ Y) \tag{Psi}$$

Interdefinability of Combinators In [CF58] it is shown that all combinators defined above can be defined in terms of **S** and **K** alone. We want our combinators to behave efficiently as a TRS, and therefore we do not take this approach.

However, we can take advantage of this fact by using the defining rules in reverse; instead of defining **I** as **SKK** we add a rule that reduces **SKK** to **I**. In this way combinators will reduce to as short a representation as possible.

We will see the advantage of this in Section 4.7, where we will translate λ-abstractions to combinators using only **I**, **S**, **K** and Φ. The length of the translated terms will decrease significantly in the presence of the following equations.

$$
\begin{array}{llll}
(1)\ \mathbf{S}\ \mathbf{K}\ M & = \mathbf{I} & (2)\ \mathbf{S}\ (\mathbf{K}\ M)\ (\mathbf{K}\ N) & = \mathbf{K}\ (M\ N) \\
(3)\ \mathbf{S}\ (\mathbf{K}\ M) & = \mathbf{B}\ M & (4)\ \mathbf{S}\ M\ (\mathbf{K}\ N) & = \mathbf{C}\ M\ N \\
(5)\ \mathbf{S}\ F\ \mathbf{I} & = \mathbf{W}\ F & (6)\ \mathbf{B}\ M\ (\mathbf{K}\ N) & = \mathbf{K}\ (M\ N) \\
(7)\ \mathbf{C}\ (\mathbf{K}\ M)\ N & = \mathbf{K}\ (M\ N) & & \\
(8)\ \mathbf{B}\ \mathbf{I}\ F & = F & (9)\ \mathbf{B}\ F\ \mathbf{I} & = F \\
(10)\ \mathbf{B}\ (\mathbf{B}\ \mathbf{S})\ \mathbf{B} & = \Phi & & \\
(11)\ \Phi\ F\ (\mathbf{K}\ M)\ (\mathbf{K}\ N) & = \mathbf{K}\ (F\ M\ N) & & \\
(12)\ \Phi\ F\ (\mathbf{K}\ M) & = \mathbf{B}\ (F\ M) & (13)\ \Phi\ F\ M\ (\mathbf{K}\ N) & = \mathbf{C}\ (\mathbf{B}\ F\ M)\ N
\end{array}
$$

Proposition 4.2.1 *(i). The left-hand side and right-hand side of each of the equations (1) until (13) are extensionally equal in the specification without these equations.*

(ii). In the presence of these equations the rewrite system remains confluent.

Proof. We show (i) and (ii) simultaneously. For each equation we show that the left-hand side and right-hand sides have one common reduct when applied to enough (arbitrary) arguments. In the reductions below the left-hand side is the left-most term and the right-hand side is the right-most term.

(1) $\mathbf{SK}MN \to \mathbf{K}N(MN) \to N \leftarrow \mathbf{I}N$

(2) $\mathbf{S}(\mathbf{K}M)(\mathbf{K}N)Y \to \mathbf{K}MY(\mathbf{K}NY) \to M(\mathbf{K}NY) \to MN \leftarrow \mathbf{K}(MN)Y$

(3) $\mathbf{S}(\mathbf{K}M)NY \to \mathbf{K}MY(NY) \to M(NY) \leftarrow \mathbf{B}MNY$

(4) $\mathbf{S}M(\mathbf{K}N)Y \to MY(\mathbf{K}NY) \to MYN \leftarrow \mathbf{C}MNY$

(5) $\mathbf{S}F\mathbf{I}X \to FX(\mathbf{I}X) \to FXX \leftarrow \mathbf{W}FX$

(6) $\mathbf{B}M(\mathbf{K}N)Y \to M(\mathbf{K}NY) \to MN \leftarrow \mathbf{K}MNY$

(7) $\mathbf{C}(\mathbf{K}M)NY \to \mathbf{K}MYN \to MN \leftarrow \mathbf{K}MNY$

(8) $\mathbf{BI}FY \to \mathbf{I}(FY) \to FY \equiv FY$

(9) $\mathbf{B}F\mathbf{I}Y \to F(\mathbf{I}Y) \to FY \equiv FY$

(10)
$$
\begin{array}{ll}
\mathbf{B}(\mathbf{BS})\mathbf{B}FMNY & \to \mathbf{BS}(\mathbf{B}F)MNY \\
& \to \mathbf{S}(\mathbf{B}FM)NY \\
& \to \mathbf{B}FMY(NY) \\
& \to F(MY)(NY) \\
& \leftarrow \Phi FMNY
\end{array}
$$

(11)
$$
\begin{array}{ll}
\Phi F(\mathbf{K}M)(\mathbf{K}N)Y & \to F(\mathbf{K}MY)(\mathbf{K}NY) \\
& \to FM(\mathbf{K}NY) \\
& \to FMN \\
& \leftarrow \mathbf{K}(FMN)Y
\end{array}
$$

$$
\begin{array}{lll}
(12) & \Phi F(\mathbf{K}M)NY & \to F(\mathbf{K}MY)(NY) \\
 & & \to FM(NY) \\
 & & \leftarrow \mathbf{B}(FM)NY \\
(13) & \Phi FM(\mathbf{K}N)Y & \to F(MY)(\mathbf{K}NY) \\
 & & \to F(MY)N \\
 & & \to \mathbf{B}FMYN \\
 & & \leftarrow \mathbf{C}(\mathbf{B}FM)NY
\end{array}
$$

■

Remark By inspecting the reductions in the proof of Proposition 4.2.1 we see that no extra reduction steps are done as a result of the added equations. Moreover, the number of reduction steps will decrease significantly if a term, containing subterms that can be reduced with one of these rules, is applied more than once.

We have now defined the basic combinators of Combinatory Logic. These can be used to represent any function or object. For instance, in [CF58] these combinators are used to define arithmetic (on naturals) and pairs.

We will define combinators of general use as new primitives. Because using only these basic combinators leads to unintelligible definitions and an inefficient TRS. However, we will later on define a mechanism to compute combinators built from the **CL** combinators: λ-abstraction.

4.3 Composition and Currying

Imports Combinatory-Logic$^{(4.2)}$

Now we define composition of functions and the transformation of n-ary functions to unary ones (uncurrying).

Exports

Context-free Functions

"$\circ$"	$\to$ Infix-Operator	{**type:** $(\beta \to \gamma) \times (\alpha \to \beta) \to \alpha \to \gamma$}
";"	$\to$ Infix-Operator	{**type:** $(\alpha \to \beta) \times (\beta \to \gamma) \to \alpha \to \gamma$}
uncurry	$\to$ Combinator	{**type:** $(\uparrow (\Gamma +\!\!+ \{\beta\}) \to \prod \Gamma \to \beta$}
curry	$\to$ Combinator	{**type:** $(\alpha \times \beta \to \gamma) \to \alpha \to \beta \to \gamma$}
"**A**"	$\to$ Combinator	{**type:** $((\prod \alpha_{\mathrm{i}} \to \beta) \times \prod \alpha_{\mathrm{i}}) \to \beta$}
"$\mathbf{A}_2$"	$\to$ Combinator	{**type:** $\alpha \to (\alpha \to \beta) \to \beta$}

Hiddens

Variables

[*FGXY*] $\to$ Term
[*X*] "*" $\to$ {Term "," }*
[*XY*] "+" $\to$ {Term "," }+

Equations

Functional Composition The composition combinator **B** is usually written as the infix operator ($\circ$).

$$F \circ G = \mathbf{B}\ F\ G$$

Another notion of the composition of two functions applies the second function after the first. We will call this sequential composition and denote it by the infix operator (;).

$$F\ ;\ G = \mathbf{B}\ G\ F$$

Example Compare the effect of the compostion operators in the composition of 'plus 1' and 'times 2' applied to 3;[1] the normal composition evaluates to

$$((+1) \circ (\cdot 2))3 \rightarrow \mathbf{B}(+1)(\cdot 2)3 \twoheadrightarrow ((3 \cdot 2) + 1) \twoheadrightarrow 7 ,$$

while the sequential composition reduces to

$$((+1)\ ;\ (\cdot 2))3 \rightarrow \mathbf{B}(\cdot 2)(+1)3 \twoheadrightarrow ((3 + 1) \cdot 2) \twoheadrightarrow 8 .$$

Curry and Uncurry The combinator *uncurry* turns a function that expects $n \geq 0$ arguments into a function that takes one n-tuple.

$$\begin{aligned} \mathit{uncurry}\ F\ \langle\rangle &= F \\ \mathit{uncurry}\ F\ \langle X, X{*}\rangle &= \mathit{uncurry}\ (F\ X)\ \langle X{*}\rangle \end{aligned}$$

For functions with 2 arguments we define the combinator *curry* that turns a binary function F (that expects a pair as its first argument) into a curried function.

$$\mathit{curry}\ F\ X\ Y = F\ \langle X, Y\rangle$$

To make functions that expect an n-tuple as first argument *curry* must have more information (the amount n).

Apply The combinator **A** (pronounce *apply*) applies the first (functional) item of a tuple to the other items in the tuple.

$$\mathbf{A}\ \langle F, X{*}\rangle = \mathit{uncurry}\ F\ \langle X{*}\rangle$$

$\mathbf{A}_2$ (pronounce *apply to*) applies its second argument to its first. $\mathbf{A}_2$ is useful when a number of functions (in a list for instance) have to be applied to the same argument.

$$\mathbf{A}_2\ X\ F = F\ X$$

$\mathbf{A}_2$ is an abbreviation for **C I**.

$$\mathbf{C\ I} = \mathbf{A}_2$$

Remark The combinator $\mathbf{A}_2$ explains the notion of 'message passing' in object-oriented programming; an object O is turned into a function ($\mathbf{A}_2\ O$) that can receive 'messages' F of type $(\alpha \rightarrow \beta)$, that is functions. The effect of sending the message is application of the function to the object (FO). If a change in the global environment is desired, the messages should have an extra argument that must be filled in with the environment to be changed (FOE).

4.4 Boolean Combinators

Imports Composition-and-Currying$^{(4.3)}$

Now we define the sort Boolean, the Boolean constants $\top$ (top or true) and $\bot$ (bottom or false) and the boolean combinators $(\neg)$ (negation), $(\wedge)$ (conjunction), $(\vee)$ (disjunction), $(\rightarrow)$ (implication) and $(\leftrightarrow)$ (equivalence). We will denote sort Boolean by $\mathbb{B}$ in types. Furthermore we define $(/)$ which turns a unary predicate into its negation, **if** for conditional choice and the equality predicate $(\equiv)$.

Exports

Sorts Boolean

[1] Arithmetic combinators will be introduced in Section 4.5

Context-free Functions

"$\top$"	$\to$ Boolean	
"$\bot$"	$\to$ Boolean	
Boolean	$\to$ Constant	
"$\neg$"	$\to$ Combinator	{**type:** $\mathbb{B} \to \mathbb{B}$}
"$\wedge$"	$\to$ Infix-Operator	{**type:** $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$}
"$\vee$"	$\to$ Infix-Operator	{**type:** $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$}
"$\to$"	$\to$ Infix-Operator	{**type:** $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$}
"$\leftrightarrow$"	$\to$ Infix-Operator	{**type:** $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$}
"/"	$\to$ Combinator	{**type:** $(\alpha \to \mathbb{B}) \to \alpha \to \mathbb{B}$}
"***if***"	$\to$ Combinator	{**type:** $\mathbb{B} \to \alpha \to \alpha \to \alpha$}
"***if***^{c}"	$\to$ Combinator	{**type:** $\mathbb{B} \times \alpha \times \alpha \to \alpha$}
"$\equiv$"	$\to$ Infix-Operator	{**type:** $\alpha \times \alpha \to \mathbb{B}$}

Variables

"$\mathcal{B}$" [*0-9′*]∗ $\to$ Boolean

Hiddens

Variables

[*BFPXY*] [*0-9′*]∗	$\to$ Term
[*XY*] [∗]	$\to$ {Term "," }∗
[*XY*] [+]	$\to$ {Term "," }+

Equations

Truth tables for ($\neg$), ($\wedge$), ($\vee$), ($\to$) and ($\leftrightarrow$).

$$\neg \top = \bot \qquad \neg \bot = \top$$

$$\begin{array}{lll} \top \wedge B = B & \top \vee B = \top & \top \to B = B \\ B \wedge \top = B & B \vee \top = \top & B \to \top = \top \\ \bot \wedge B = \bot & \bot \vee B = B & \bot \to B = \top \\ B \wedge \bot = \bot & B \vee \bot = B & B \to \bot = \neg B \end{array}$$

$$B_1 \leftrightarrow B_2 = B_1 \to B_2 \wedge (B_2 \to B_1)$$

Note that the operations are defined on other Terms than Booleans provided that one of the operands is a Boolean. This means that we can abuse these operations for other purposes. Conjunction can be used to define a 'guarded' operation that yields an expression B if some boolean expression evaluates to $\top$ and that yields $\bot$ if the boolean expression evaluates to $\bot$.

Neg (/) turns a unary predicate into its negation. (The notation / hints at the stroke through a relation symbol to denote its complement as in $\not\geq$.)

$$/ \; P \; X = \neg \; (P \; X)$$

Remark / equals ($\mathbf{B}\neg$). The complements of binary, ternary, and n-ary predicates are, respectively, ($\mathbf{B}(\mathbf{B}\neg)$), ($\mathbf{B}(\mathbf{B}(\mathbf{B}\neg))$), ($\mathbf{B}^n\neg$). (The definition of $.^n$ will be given in 4.8.)

If is conditional choice:

$$\begin{array}{llll} \textit{\textbf{if}}\,\top\ X\ Y & = X & \textit{\textbf{if}}\,\bot\ X\ Y = Y \\ \textit{\textbf{if}}\,B\ (F\ X)\ (F\ Y) & = F\ (\textit{\textbf{if}}\,B\ X\ Y) & \textit{\textbf{if}}\,B\ X\ X = X \end{array}$$

(See [Hee92b] for the last two equations.)

With *uncurry* we can define the uncurried if-then-else. The combinator **if**^{c} operates on a triple of arguments; the first is the conditional and the second and third the then and else branches, respectively.

$$\textit{\textbf{if}}^c = \textit{uncurry}\ \textit{\textbf{if}}$$

Example With the aritmethic combinators from Section 4.5 we have

$$\mathbf{if}^c\langle 2 > 3, 10, -10\rangle \twoheadrightarrow -10$$

Equality The predicate $(\equiv)$ decides whether two terms are equal. Two constants are equal if they have the same normal form. We first state that any constant δ is equal to itself.

$$\delta \equiv \delta = \top$$

Then, if $\delta 1$ and $\delta 2$ reduce to different normal forms ($\delta 1 \neq \delta 2$) they are unequal. (But to be a true constant, a constant should always be in normal form.)

$$\delta_1 \equiv \delta_2 = \bot \Leftarrow \delta_1 \neq \delta_2$$

Example

$$1 + 1 \equiv 2 \twoheadrightarrow 2 \equiv 2 \rightarrow \top$$

If one of the arguments of $(\equiv)$ is a constant and the other is a list or a tuple they are not equal.

$$\begin{array}{ll} \delta \equiv [Y*] = \bot & \delta \equiv \langle X, X+\rangle = \bot \\ [X*] \equiv \delta = \bot & \langle X, X+\rangle \equiv \delta = \bot \end{array}$$

Note that we did not define equality of applications and infix-operation terms in a generic manner, as for constants. The reason for this is the danger of inconsistency. The equations above do not lead to inconsistency if no equations are defined over constants, lists and tuples. However, if we would add the rule

$$(M\ \odot\ N) \equiv \delta = \bot \qquad \bigstar$$

then we would have

$$1 + 1 \equiv 2 \rightarrow \bot$$

with the conclusion that $(\top = \bot)$.[2]

Equality of lists and tuples will be defined in Section 5.6.

4.5 Combinatory Arithmetic

Imports Naturals$^{(??)}$ Boolean-Combinators$^{(4.4)}$

We import a module Naturals which defines the sorts NAT_CON (natural constants), NAT (first-order expressions over NAT_CON), and BOOL (booleans; for

[2]The symbol $\bigstar$ will be used to indicate that an equation is not part of the specification but only used to illustrate some point.

results of comparisons). The constants are added as combinatory constants. Arithmetic combinators, such as + and −, are defined in terms of NAT expressions.

We do not show module Naturals itself. It may be any module that defines the sorts above with the obvious semantics for the functions. Examples of specifications containing such modules appear in [Deu92b].

In type annotations we will write $\mathbb{N}$ for NAT_CON.

Exports

Context-free Functions

NAT_CON → Constant {**type:** $\mathbb{N}$}

"+" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$}
"−" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$}
"·" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$}

"<" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$}
">" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$}
"≤" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$}
"≥" → Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$}

"*digit?*" → Combinator {**type:** $\mathbb{N} \to \mathbb{B}$}

Variables

"*n*" [*0-9'*]* → NAT_CON
"*i*" [*0-9'*]* → NAT_CON
"*j*" [*0-9'*]* → NAT_CON
"*k*" [*0-9'*]* → NAT_CON
"*l*" [*0-9'*]* → NAT_CON
"*m*" [*0-9'*]* → NAT_CON
"$\vec{n}$" [*0-9'*]* → {Term "," }*

"n^+" [*0-9'*]* → {NAT "," }+

The implementation of the arithmetic combinators above is caried out by putting brackets around the expression over sort NAT (or BOOL in case of comparison operaterors) so that the expression can be computed by the equations in module Naturals. When the computation yields a NAT_CON (or Boolean constant) the brackets are removed.

Hiddens

Context-free Functions

"⟦" NAT "⟧" → Constant {**type:** $\mathbb{N}$}
"⟦" BOOL "⟧" → Boolean {**type:** $\mathbb{B}$}

Equations

Brackets Semantics of ⟦BOOL⟧ and ⟦NAT⟧.

$$[\![true]\!] = \top \qquad [\![false]\!] = \bot \qquad [\![n]\!] = n$$

Arithmetic combinators are implemented by their NAT equivalents. The combinators are only defined on NAT_CON's. This entails they can be overloaded for use on other sorts.

$$i + j = [\![i + j]\!] \qquad i - j = [\![i - j]\!] \qquad i \cdot j = [\![i * j]\!]$$

Comparison of naturals.

$$i < j = [\![i < j]\!] \qquad i > j = [\![i > j]\!]$$
$$i \leq j = [\![i <= j]\!] \qquad i \geq j = [\![i >= j]\!]$$

Note that the operators in the left-hand sides are Infix-Operator's while on the right-hand side they are NAT operators.

digit? decides whether its arguments is a natural between 0 and 9, i.e. a digit.

$$digit?\ n = 0 \leq n \wedge (n \leq 9)$$

Remark It is instructive to see that hiding of implementation was already an issue in [CHS72].

Curry, Feys and Seldin devote a whole chapter (13) to Combinatory Arithmetic. The arithmetic combinators are there defined in pure **CL** (in terms of **S** and **K**). Since there are many ways to define 0 and (+1) using combinators they choose at some point (p. 214) to define new symbols that denote zero and the successor function and that can be 'implemented' by any combinators that will do the job. They say the following about this.

> These new primitives will be symbolized as $[\![0]\!]$ and $[\![\sigma]\!]$ respectively. Then the representative $[\![n]\!]$ of the natural number n will be defined recursively, since it has already been defined for $n = 0$, by the stipulation
>
> $$[\![n']\!] \equiv [\![\sigma]\!][\![n]\!]$$
>
> [...]
>
> This procedure has certain advantages besides generality. To those who, like Kronecker, regard the natural numbers as acts of God, which it is irreverent to define in terms of anything else, as well as to others who prefer to regard the natural numbers as primitive for less esoteric reasons, it has a certain philosophical advantage.

4.6 Lists and Tuples

Imports Combinatory-Arithmetic$^{(4.5)}$

Any construction can be considered as a list, if at least the operations *hd*, *tl*, *empty?*, *nil?* and (:) are defined on it. All list operations defined in chapter 5 are applicable to lists defined in this way.

Apart from the definition of these primitive list operations we introduce primitive tuple operators and some additional operations on lists and tuples.

Exports

Context-free Functions

nil	→ Combinator	{**type:** $\alpha^* \to \alpha^*$}
":"	→ Right-Infix-Operator	{**type:** $\alpha \times \alpha^* \to \alpha^+$}
hd	→ Combinator	{**type:** $\alpha^+ \to \alpha$}
tl	→ Combinator	{**type:** $\alpha^+ \to \alpha^*$}
"ε?"	→ Combinator	{**type:** $\alpha^* \to \mathbb{B}$}
"<:"	→ Infix-Operator	{**type:** $\alpha^* \to \alpha \to \alpha^+$}
init	→ Combinator	{**type:** $\alpha^+ \to \alpha^*$}
last	→ Combinator	{**type:** $\alpha^+ \to \alpha$}
"**P**"	→ Combinator	{**type:** $\alpha \to \beta \to \alpha \times \beta$}
"$\mathbf{P}_n$"	→ Combinator	{**type:** $\alpha_0 \to \prod_1^{\mathrm{n}} \alpha_{\mathrm{i}} \to \prod_0^{\mathrm{n}} \alpha_{\mathrm{i}}$}
"π"	→ Combinator	{**type:** $(\mathbb{N} \to \alpha^+ \to \alpha) \cup (\mathbb{N} \to \prod \alpha_{\mathrm{i}} \to \alpha_{\mathrm{i}})$}
"*set*"	→ Combinator	{**type:** $(\mathbb{N} \to \alpha \to \alpha^+ \to \alpha^+)$ $\cup(\mathbb{N} \to \alpha_{\mathrm{i}} \to \prod \alpha_{\mathrm{i}} \to \prod \alpha_{\mathrm{i}})$}

Hiddens

Variables

$[LXY] \rightarrow$ Term
$[X]$ "$*$" $\rightarrow$ {Term "," }$*$

Equations

Primitive List Combinators

The ADT list consists of the combinators *nil*, (:) (cons), *hd* (head), *tl* (tail) and $\varepsilon?$ (empty?). Any structure can be defined as a list when these five operations are defined on it; all other list operations are defined in terms of these four combinators.

Requirements Any specification of the primitive list operations should respect the following equations. The combinator : (*cons*) adds an element at the front of a list. *Hd* gives the first element (head) of a list, *tl* the rest (tail) of the list. Lists must have at least one element.

$$hd\,(X : L) = X \qquad tl\,(X : L) = L$$

$\varepsilon?$ yields $\top$ if its argument is an empty list, $\perp$ otherwise.

$$\varepsilon?\ (nil\,L) = \top \qquad \varepsilon?\ (X : L) = \perp$$

Given a list, *nil* yields the 'empty list' associated with that list.

$$nil\,L = L \Leftarrow \varepsilon?\ L = \top$$

Furthermore *hd* and *tl* should be defined on any list L for which $(\varepsilon?\ L)$ is $\perp$ and in that case $(hd\,L : tl\,L)$ should be equal to L.

[]-lists Now we give the definition of the primitive list operations on []-lists. (Henceforth we will use '[]-list' to speak about lists constructed by a list of Term's in square brackets, and 'list' to denote any structure for which the primitive list operations have been defined.)

$$\begin{array}{llll} nil\,[X*] & = [] & X : [X*] & = [X, X*] \\ hd\,[X, X*] & = X & tl\,[X, X*] & = [X*] \\ \varepsilon?\ [] & = \top & \varepsilon?\ [X, X*] & = \perp \end{array}$$

Some Additional List Combinators

Init and Last The last element, respectively, the initial segment of a list (the anologues of *hd* and *tl* with respect to the last element) are given by *last* and *init*

$$last\,[X*, X] = X \qquad init\,[X*, X] = [X*]$$

And for other lists.

$$\begin{array}{ll} last\ L = hd\ L & \Leftarrow \varepsilon?\ L = \perp,\ \varepsilon?\ (tl\,L) = \top \\ last\ L = last\,(tl\,L) & \Leftarrow \varepsilon?\ L = \perp,\ \varepsilon?\ (tl\,L) = \perp \\ init\ L = nil\ L & \Leftarrow \varepsilon?\ L = \perp,\ \varepsilon?\ (tl\,L) = \top \\ init\ L = hd\ L : init\,(tl\,L) & \Leftarrow \varepsilon?\ L = \perp,\ \varepsilon?\ (tl\,L) = \perp \end{array}$$

Cons-Last ($<:$) adds an element at the end of a list.

$$
\begin{array}{lll}
[X*] <: X = [X*, X] & & \\
L <: X & = X : L & \Leftarrow \varepsilon?\ L = \top \\
L <: X & = \mathit{hd}\ L : \mathit{tl}\ L <: X & \Leftarrow \varepsilon?\ L = \bot
\end{array}
$$

Primitive Tuple Combinators

Tuples are sequences of items between angle brackets '$\langle$' and '$\rangle$'. We provide operations to construct and decompose tuples. The combinator **P** forms a pair of its two arguments, The combinator $\mathbf{P}_n$ adds one item to a tuple.

$$\mathbf{P}\ X\ Y = \langle X,\ Y\rangle \qquad \mathbf{P}_n\ X\ \langle X*\rangle = \langle X,\ X*\rangle$$

Tuples have the same syntactic structure as lists (modulo the brackets), therefore we can define the primitive list operations for tuples as well. Note however, that tuples have not the same type as lists. Defining the primitive list operations for tuples entails overloading these operations.

$$
\begin{array}{llll}
\mathit{hd}\ \langle X,\ X*\rangle & = X & \mathit{tl}\ \langle X,\ X*\rangle & = \langle X*\rangle \\
\varepsilon?\ \langle\rangle & = \top & \varepsilon?\ \langle X,\ X*\rangle & = \bot \\
\mathit{nil}\ \langle X*\rangle & = \langle\rangle & X : \langle X*\rangle & = \langle X,\ X*\rangle
\end{array}
$$

We see here that tuples can be treated like lists. The difference between tuples and lists is their type. Lists have type α^* while tuples have type $(\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n)$ (with $n \geq 0$).

Projection The combinator π takes a natural n and a tuple or a list and yields the n-th element of that tuple or list (counting from 1).

$$
\begin{array}{ll}
\pi\ 1\ L = \mathit{hd}\ L & \Leftarrow \varepsilon?\ L = \bot \\
\pi\ n\ L = \pi\ (n-1)\ (\mathit{tl}\ L) & \Leftarrow \varepsilon?\ L = \bot,\ \ n > 1 = \top
\end{array}
$$

Set changes the n-th element of a tuple or list L to Y, leaving the rest unchanged.

$$
\begin{array}{ll}
\mathit{set}\ 1\ \ Y\ L = Y : \mathit{tl}\ L & \Leftarrow \varepsilon?\ L = \bot \\
\mathit{set}\ n\ \ Y\ L = \mathit{hd}\ L : \mathit{set}\ (n-1)\ \ Y\ (\mathit{tl}\ L) & \Leftarrow n > 1 = \top,\ \ \varepsilon?\ L = \bot
\end{array}
$$

So we have, if Y is a list or tuple with $\geq n$ elements,

$$\pi\ n\ (\mathit{set}\ n\ X\ Y) = X$$

4.7 Functional Abstraction

Imports Lists-and-Tuples$^{(4.6)}$

We introduce free variables and a new syntactic construct: λ-abstraction. The semantics of this construct is defined in terms of the **CL** combinators of Section 4.2. The λ notation serves as abbreviation for complicated combinators.

Exports
  Sorts Variable
  Lexical Syntax
    "_" *[a-zA-Z]** *[0-9]** $\rightarrow$ Variable
  Context-free Functions

Variable → Term {**type:** α}
"λ" Variable+ "." Term → Term {**type:** $\frac{\text{type(Variable}^+\text{)}=\Gamma,\text{type(Term}_1\text{)}=\beta}{\text{type(Term}_2\text{)}=\uparrow\Gamma\cup\beta}$}

Variables
"$\mathcal{V}$" [*0-9'*]* → Variable
"$\mathcal{V}^+$" [*0-9'*]* → Variable+

Priorities
"λ" Variable+ "." Term → Term <
{ Term Term → Term, Term Infix-Operator Term → Term,
Term Right-Infix-Operator Term → Term, Term Postfix-Operator → Term }

We will say that *a variable* $\mathcal{V}$ *is free in a* Term M if $\mathcal{V}$ is not a subterm of M. We will write $M[\mathcal{V} := N]$ for the substitution of N for each occurrence of $\mathcal{V}$ in M.

Hiddens

Variables
[*FGHMNXY*] → Term
[*X*] "*" → {Term "," }*
[*XY*] "+" → {Term "," }+

Equations

In the λ-calculus the semantics of the construct $(\lambda\mathcal{V}.M)$ is defined by the β-reduction rule as

$$(\lambda\mathcal{V}.M)N \to_{(\beta)} M[\mathcal{V} := N]\ .$$

This definition implies that β-reduction implements substition of terms for variables; not that we need substition to implement β-abstraction.

Since substitution is an expensive operation, involving renaming of variables (α-conversion), we translate terms containing λ's to a combinatory term without λ's. This is done by the classical algorithm used to show the equivalence of Combinatory Logic and the λ-calculus [CF58, Bar84].

The combinatory term F resulting from the abstraction $(\lambda\mathcal{V}.M)$ is a function that, when applied to an argument N, yields $M[\mathcal{V} := M]$ without doing any substitutions.

The Algorithm First we define $(\lambda\_x\_y \ldots .M)$ to be an abbreviation of $(\lambda\_x.(\lambda\_y \ldots .M))$.

$$\lambda\ \mathcal{V}\ \mathcal{V}^+\ .\ M = \lambda\ \mathcal{V}\ .\ \lambda\ \mathcal{V}^+\ .\ M \qquad \text{(L-Vs)}$$

Then we translate a term containing λ's to an equivalent term without λ's.

Atomic Terms The simple cases are abstracting a variable from an atomic term. $(\lambda\mathcal{V}.\mathcal{V})$ is the identity function.

$$\lambda\ \mathcal{V}\ .\ \mathcal{V} = \mathbf{I} \qquad \text{(L-I)}$$

$(\lambda\mathcal{V}.\mathcal{V}')$ is the constant function that yields $\mathcal{V}'$ when applied to any argument, when $\mathcal{V}$ and $\mathcal{V}'$ are different variables.

$$\lambda\ \mathcal{V}\ .\ \mathcal{V}' = \mathbf{K}\ \mathcal{V}' \Leftarrow \mathcal{V}' \neq \mathcal{V} \qquad \text{(L-Var)}$$

Constants are atomic as far as λ is concerned. A constant δ is some syntactic structure outside the syntax of combinatory terms, therefore it can not contain variables.

$$\lambda\ \mathcal{V}\ .\ \delta = \mathbf{K}\ \delta \qquad \text{(L-Const)}$$

Combinators (including combinatorized infix and postfix operators) are atomic.

$$\lambda\, \mathcal{V}\, .\, \Box = \mathbf{K}\, \Box \tag{L-Comb}$$

Remark When some new syntax over combinatory terms, that can contain Variables, is introduced λ abstraction should be defined over that syntax. Either the syntax is normalized to the syntax in this module (as is done with list comprehensions in Section 5.9) or the algorithm presented here is extended to include the new syntax.

Application When abstracting a term (MN) we have to transform it into a function that takes an argument and transports it to the place where the variable is. This is done using combinator **S** as follows:

$$\lambda\, \mathcal{V}\, .\, M\, N = \mathbf{S}\, (\lambda\, \mathcal{V}\, .\, M)\, (\lambda\, \mathcal{V}\, .\, N) \tag{L-A}$$

Applying this to a term X will yield $(\lambda\mathcal{V}.M)\ X\ ((\lambda\mathcal{V}.N)\ X)$.

[]-lists The empty []-list is atomic, since it does not contain variables. Therefore

$$\lambda\, \mathcal{V}\, .\, [] = \mathbf{K}\, [] \tag{L-L0}$$

To abstract a non-empty list $[X, X*]$ we first note that it is equal to $X : [X*]$ by the rules for (:) in Section 4.6. Now we abstract $\mathcal{V}$ from X and $[X*]$ yielding functions F and G, respectively. We have to construct a function that yields $(FX : GX)$ when applied to a term X. This is done by $\Phi(:)$.

$$\lambda\, \mathcal{V}\, .\, [X,\, X*] = \Phi\, (:)\, (\lambda\, \mathcal{V}\, .\, X)\, (\lambda\, \mathcal{V}\, .\, [X*]) \tag{L-Ln}$$

⟨⟩-tuples The functional abstraction of *tuples* is similar to the list case; the combination $\Phi\mathbf{P}_n$ reconstructs the tuple after application.

$$\lambda\, \mathcal{V}\, .\, \langle\rangle = \mathbf{K}\, \langle\rangle \tag{L-T0}$$

$$\lambda\, \mathcal{V}\, .\, \langle X,\, X*\rangle = \Phi\, \mathbf{P}_n\, (\lambda\, \mathcal{V}\, .\, X)\, (\lambda\, \mathcal{V}\, .\, \langle X*\rangle) \tag{L-Tn}$$

Infix and Postfix Operators Abstraction of infix and postfix operators is similar to application. However, instead of using **S** to reconstruct the application we use $(\Phi\ (\odot))$.

$$\lambda\, \mathcal{V}\, .\, M \odot N = \Phi\, (\odot)\, (\lambda\, \mathcal{V}\, .\, M)\, (\lambda\, \mathcal{V}\, .\, N) \tag{L-Inf-1}$$

$$\lambda\, \mathcal{V}\, .\, M \circledcirc N = \Phi\, (\circledcirc)\, (\lambda\, \mathcal{V}\, .\, M)\, (\lambda\, \mathcal{V}\, .\, N) \tag{L-RInf-1}$$

To abstract left sectioned infix operators, we first note that $(M\odot)$ is equal to $(\odot)M$. Since $(\odot)$ is a combinator the abstraction $(\lambda\mathcal{V}.(\odot))$ is equal to $\mathbf{K}(\odot)$. By the rule for application (L-A) we see that $(\lambda\mathcal{V}.((\odot)\ M))$ is $(\mathbf{S}\ (\mathbf{K}(\odot))\ (\lambda\mathcal{V}.M))$. The prefix of this term is equal to $(\mathbf{B}\ (\odot))$. Therefore

$$\lambda\, \mathcal{V}\, .\, (M \odot) = \mathbf{B}\, (\odot)\, (\lambda\, \mathcal{V}\, .\, M) \tag{L-Inf-2}$$

$$\lambda\, \mathcal{V}\, .\, (M \circledcirc) = \mathbf{B}\, (\circledcirc)\, (\lambda\, \mathcal{V}\, .\, M) \tag{L-RInf-2}$$

Right sectioning is similar to left sectioning after we note that $(\odot\ N)$ is equal to $(\mathbf{C}\ (\odot)\ N)$.

$$\lambda\, \mathcal{V}\, .\, (\odot\, N) = \mathbf{B}\, (\mathbf{C}\, (\odot))\, (\lambda\, \mathcal{V}\, .\, N) \tag{L-Inf-3}$$

$$\lambda\, \mathcal{V}\, .\, (\circledcirc\, N) = \mathbf{B}\, (\mathbf{C}\, (\circledcirc))\, (\lambda\, \mathcal{V}\, .\, N) \tag{L-RInf-3}$$

Application of a postfix operator $(M\ \boxdot)$ is equal to $((\boxdot)\ M)$.

$$\lambda\, \mathcal{V}\, .\, M \boxdot = \mathbf{B}\, (\boxdot)\, (\lambda\, \mathcal{V}\, .\, M) \tag{L-Pfix}$$

Remark Note that we did not have to face the situation $(\lambda\mathcal{V}.\lambda\mathcal{V}'.M)$, which is problematic when functional abstraction is defined in terms of substitution. Here it does not pose a problem because the inner abstraction will reduce to a term not containing $\mathcal{V}'$.

The following proposition states that the algorithm above is correct.

Proposition 4.7.1 (Correctness) *For any terms M and N and variable $\mathcal{V}$*

$$(\lambda\mathcal{V}.M)N \twoheadrightarrow M[\mathcal{V} := N]$$

Proof. The proof is by a straightforward induction on the structure of M. We do the step for Infix-Operator's; if $M \equiv (P \odot Q)$ then

$$\begin{aligned}
(\lambda\mathcal{V}.(P \odot Q))N &\rightarrow \Phi(\odot)(\lambda\mathcal{V}.P)(\lambda\mathcal{V}.Q)N && \text{(L-Inf-1)}\\
&\rightarrow (\odot)((\lambda\mathcal{V}.P)N)((\lambda\mathcal{V}.Q)N) && \text{(4.2.Phi)}\\
&\rightarrow (\lambda\mathcal{V}.P)N \odot (\lambda\mathcal{V}.Q)N && \text{(3.1.1)}\\
&\twoheadrightarrow P[\mathcal{V} := N] \odot Q[\mathcal{V} := N] && \text{i.h.}\\
&\equiv (P \odot Q)[\mathcal{V} := N] && \text{subst}
\end{aligned}$$

■

The Result of Abstractions The transformation above can yield rather complex combinatory terms. For instance $(\lambda\_x\_y\_z.(\_x\_z(\_y\_z)))$ reduces to

$$\begin{aligned}
&\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KS})))\,(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KS})))\\
&(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KK})))\,(\mathbf{S}(\mathbf{KK})\mathbf{I}))))\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KI})))))\,(\mathbf{S}(\mathbf{S}(\mathbf{KS})\\
&(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KS})))\,(\mathbf{S}(\mathbf{S}(\mathbf{KS})\,(\mathbf{S}(\mathbf{KK})\,(\mathbf{KK})))\,(\mathbf{KI}))))\,(\mathbf{S}(\mathbf{KK})\\
&(\mathbf{KI})))
\end{aligned}$$

But since we defined some additional equations over the standard combinators in Section 4.2 that term reduces to $\mathbf{S}$ (as it should). Furthermore, we have

$$\lambda\_x\_y.(\_x) \twoheadrightarrow \mathbf{K}\ .$$

The following proposition states that a λ-abstraction of a term not containing the abstracted variable reduces to the smallest representation of that abstraction.

Proposition 4.7.2 *For any term M and any variable $\mathcal{V}$*

(i). $\lambda\mathcal{V}.M \twoheadrightarrow \mathbf{K}M$ *, and*

(ii). $\lambda\mathcal{V}.(M\mathcal{V}) \twoheadrightarrow M$

if $\mathcal{V}$ is not free in M.

Proof. (i) by induction on the structure of M (using equations (4.2.1) until (4.2.13)).

(ii) follows by (i):

$$\begin{aligned}
\lambda\mathcal{V}.(M\mathcal{V}) &\rightarrow \mathbf{S}(\lambda\mathcal{V}.M)(\lambda\mathcal{V}.\mathcal{V}) && \text{(L-A)}\\
&\twoheadrightarrow \mathbf{S}(\mathbf{K}M)(\lambda\mathcal{V}.\mathcal{V}) && \text{(i)}\\
&\rightarrow \mathbf{S}(\mathbf{K}M)\mathbf{I} && \text{(L-I)}\\
&\rightarrow \mathbf{B}M\mathbf{I} && \text{(4.2.3)}\\
&\rightarrow M && \text{(4.2.9)}
\end{aligned}$$

■

4.8 Recursion

Imports Combinatory-Arithmetic(4.5)

Now we define powers of functions and the primitive recursion operation.

Exports

Context-free Functions

"^" $\rightarrow$ Infix-Operator {**type:** $(\alpha \rightarrow \alpha) \times \mathbb{N} \rightarrow \alpha \rightarrow \alpha$}

prim $\rightarrow$ Combinator {**type:** $\alpha \rightarrow (\mathbb{N} \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha$}

Hiddens

Variables

[*FGHMNXY*] $\rightarrow$ Term
[*X*] "*" $\rightarrow$ {Term "," }*
[*XY*] "+" $\rightarrow$ {Term "," }+

Equations

Powers The infix operator (^) turns a function into n applications of that function.

$$\begin{aligned} F \,\hat{}\, 0 &= \mathbf{I} \\ (F \,\hat{}\, n)\; X &= (F \,\hat{}\, (n-1))\; (F\; X) \Leftarrow n > 0 = \top \end{aligned}$$

Example For instance, $((\mathbf{K} \,\hat{}\, n)\; Y)$ is the n-ary constant function; it eats n arguments and then yields Y.

$(tl \,\hat{}\, n)$ is the function that yields the n-th tail of a list of n or more items; $(tl \,\hat{}\, 2)\; [1, 2, 3, 4] = [3, 4]$.

Primitive Recursion (See [CHS72] chapter 13 p. 220)

$$\begin{aligned} &prim\; G\; H\; 0 = G \\ &prim\; G\; H\; n = H\; n'\; (prim\; G\; H\; n') \Leftarrow n > 0 = \top,\;\; n' = n - 1 \end{aligned}$$

Example The power operator defined above could be defined using *prim* as

$$F \,\hat{}\, n = prim\; \mathbf{I}\; (\mathbf{K}(F\;;)) \qquad \bigstar$$

since

$$prim\; \mathbf{I}\; (\mathbf{K}(F\;;))\; 0 = \mathbf{I} \qquad \bigstar$$

and

$$\begin{aligned} prim\; \mathbf{I}\; (\mathbf{K}(F\;;))\; n\; X &= \mathbf{K}\; (F\;;)\; (n-1)(prim\; \mathbf{I}\; (\mathbf{K}(F\;;))\; (n-1))\; X \\ &= \mathbf{K}\; (F\;;)\; (n-1)(F \,\hat{}(n-1))\; X \\ &= (F\;;\; (F \,\hat{}(n-1)))\; X \\ &= (F \,\hat{}(n-1))\; (F\; X) \\ &= (F \,\hat{}\, n)\; X \end{aligned}$$

if $n > 0$.

4.9 Characters and Strings

Imports Lists-and-Tuples[(4.6)] Map-and-Fold[(5.2)]

The syntax of constants is extended with two more sorts: Characters and Strings.

Exports

Sorts Character String

Lexical Syntax

$$\begin{array}{ll} \text{“‘”}\ \sim[\backslash n]\ \text{“’”} & \to \text{Character} \\ \text{“}\backslash\text{"”}\ \sim[\backslash n \backslash\text{"}]{*}\ \text{“}\backslash\text{"”} & \to \text{String} \end{array}$$

Context-free Functions

$$\begin{array}{ll} \text{Character} & \to \text{Constant} \\ \text{String} & \to \text{Constant} \end{array}$$

$$\begin{array}{ll} \mathit{str2cl} & \to \text{Combinator } \{\textbf{type: } \text{String} \to \text{Character}^*\} \\ \text{“}{\downarrow}_{\text{“”}}\text{”} & \to \text{Combinator } \{\textbf{type: } \text{Character}^* \to \text{String}\} \\ \mathit{n2s} & \to \text{Combinator } \{\textbf{type: } \mathbb{N} \to \text{String}\} \end{array}$$

Variables

$$\begin{array}{ll} \text{“}\mathit{Char}\text{”}\ [\mathit{0\text{-}9'}]{*} & \to \text{Character} \\ \text{“}\mathit{String}\text{”}\ [\mathit{0\text{-}9'}]{*} & \to \text{String} \end{array}$$

Hiddens

Variables

$$\begin{array}{ll} [c] & \to \text{CHAR} \\ [c]\ [*] & \to \text{CHAR}{*} \\ [c]\ [+] & \to \text{CHAR+} \end{array}$$

Equations

Strings as Lists of Characters We define strings as lists of characters by giving definitions for the primitive list operations.

$$\begin{array}{ll} \mathit{character}(\text{“‘”}\ \ c\ \text{“’”}\ ) : \mathit{string}(\text{“"”}\ \ c{*}\ \text{“"”}\ ) & = \mathit{string}(\text{“"”}\ \ c\ c{*}\ \text{“"”}\ ) \\ \mathit{hd}\ \mathit{string}(\text{“"”}\ \ c\ c{*}\ \text{“"”}\ ) & = \mathit{character}(\text{“‘”}\ \ c\ \text{“’”}\ ) \\ \mathit{tl}\ \mathit{string}(\text{“"”}\ \ c\ c{*}\ \text{“"”}\ ) & = \mathit{string}(\text{“"”}\ \ c{*}\ \text{“"”}\ ) \\ \varepsilon?\ \text{“”} & = \top \\ \varepsilon?\ \mathit{string}(\text{“"”}\ \ c\ c{*}\ \text{“"”}\ ) & = \bot \\ \mathit{nil}\ \mathit{String} & = \text{“”} \end{array}$$

Conversion of lists of characters to strings and back. We make use of the combinator $\downarrow$ that will be defined in Section 5.2. It makes a list conversion function from an empty list that indicates the type of the list.

$$\downarrow_{\text{“”}} = \downarrow \text{“”} \qquad \mathit{str2cl} = \downarrow []$$

Naturals as Lists of Characters We use $(\mathit{nil}\ n)$ as the empty digit list for some natural n (0 by default).

$$\begin{array}{ll} \mathit{character}(\text{“‘”}\ \ c\ \text{“’”}\ ) : \mathit{nil}\ n & = \mathit{nat\_con}(c) \\ \mathit{character}(\text{“‘”}\ \ c\ \text{“’”}\ ) : \mathit{nat\_con}(c+) & = \mathit{nat\_con}(c\ c+) \end{array}$$

$$\begin{array}{llll} \mathit{hd}\ \mathit{nat\_con}(c\ c{*}) & = \mathit{character}(\text{“‘”}\ \ c\ \text{“’”}\ ) & & \\ \mathit{tl}\ \mathit{nat\_con}(c\ c+) & = \mathit{nat\_con}(c+) & \mathit{tl}\ \mathit{nat\_con}(c) & = \mathit{nil}\ 0 \\ \varepsilon?\ n & = \bot & \varepsilon?\ (\mathit{nil}\ n) & = \top \end{array}$$

Naturals to strings

$$n2s\; n = \downarrow_{\text{``''}}\; n$$

In this way we can treat all lexical sorts and have at our disposal a score of operations on lexicals.

Chapter 5

List Operations

We define a number of general purpose list operations. It has been the aim to stay compatible with notation used in functional programming languages such as Miranda. Many functions are from the introduction to Miranda [BW88].

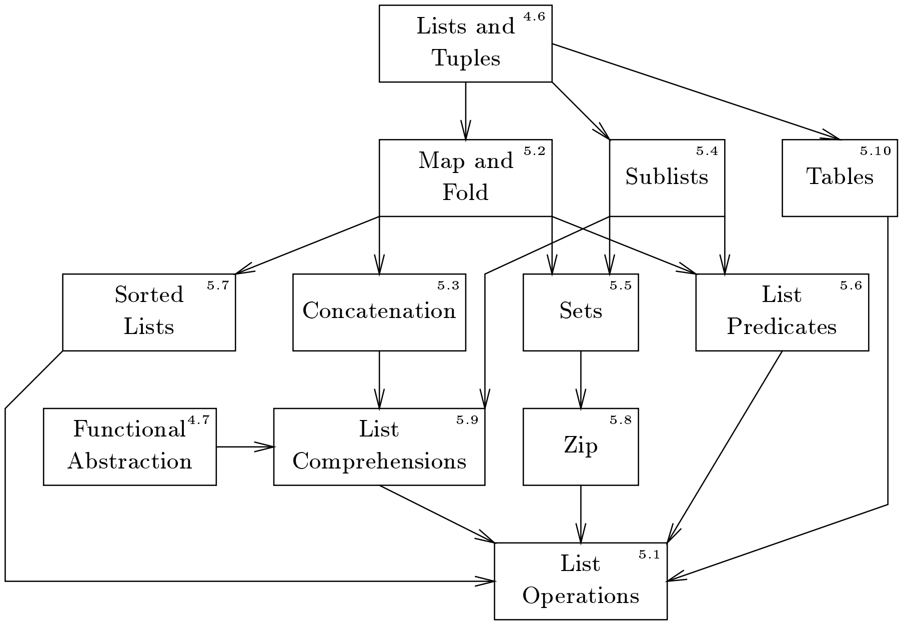


Figure 5.1: Import graph for Chapter 5; List Operations

5.1 List Operations

Imports Zip$^{(5.8)}$ List-Comprehensions$^{(5.9)}$ Sorted-Lists$^{(5.7)}$ Tables$^{(5.10)}$ List-Predicates$^{(5.6)}$

Here we collect all list operation defined in this chapter.

5.2 Map and Fold

Imports Lists-and-Tuples$^{(4.6)}$

Exports

Context-free Functions

map $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta) \rightarrow \alpha^* \rightarrow \beta^*$}
mapAcc $\rightarrow$ Combinator {**type:** $(\gamma \rightarrow \alpha \rightarrow \gamma \times \beta) \rightarrow \beta^* \rightarrow \gamma \rightarrow \alpha^* \rightarrow \gamma \times \beta^*$}
foldr $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha^* \rightarrow \beta$}
foldl $\rightarrow$ Combinator {**type:** $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow \alpha^* \rightarrow \beta$}

Exports

Context-free Functions

"$\downarrow_{[]}$" $\rightarrow$ Combinator
"$\downarrow$" $\rightarrow$ Combinator

Hiddens

Variables

[*FGHLPTXYZ*] [*0-9′*]* $\rightarrow$ Term
[*XYZ*] "*" $\rightarrow$ {Term "," }*
[*XYZ*] "+" $\rightarrow$ {Term "," }+

Equations

Map *Map* applies a function F to all members of a list yielding the list of the results.

$$\begin{aligned} map\ F\ L &= L && \Leftarrow \varepsilon?\ L = \top \\ map\ F\ L &= F\ (hd\ L) : map\ F\ (tl\ L) && \Leftarrow \varepsilon?\ L = \bot \end{aligned}$$

Additional laws for *map*.

$$map\ \mathbf{I} = \mathbf{I} \qquad \mathbf{B}\ (map\ F)\ (map\ G) = map\ (\mathbf{B}\ F\ G)$$

Foldr and Foldl Fold lists into an environment Y by a function F.

$$\begin{aligned} foldr\ F\ Y\ L &= Y && \Leftarrow \varepsilon?\ L = \top \\ foldr\ F\ Y\ L &= F\ (hd\ L)\ (foldr\ F\ Y\ (tl\ L)) && \Leftarrow \varepsilon?\ L = \bot \end{aligned}$$

$$\begin{aligned} foldl\ F\ Y\ L &= Y && \Leftarrow \varepsilon?\ L = \top \\ foldl\ F\ Y\ L &= foldl\ F\ (F\ Y\ (hd\ L))\ (tl\ L) && \Leftarrow \varepsilon?\ L = \bot \end{aligned}$$

The functions are best explained by an example.

$$foldr\ (\odot)\ Z\ [1,2,3] \twoheadrightarrow 1 \odot (2 \odot (3 \odot Z))$$

and

$$foldl\ (\odot)\ Z\ [1,2,3] \twoheadrightarrow Z \odot 1 \odot 2 \odot 3$$

List Conversion $\downarrow$ (pack) builds a conversion function for some list L.

$$\downarrow L = foldr\ (:)\ (nil\ L)$$

For instance the combinator $\downarrow_{[]}$ (unpack) converts any list to a []-list.

$$\downarrow_{[]} = \downarrow []$$

In module 4.9 we defined the conversion function to strings $\downarrow_{\text{""}}$. With this combinator we can convert a natural (which we defined as list of characters) to a list of strings.

$$\begin{aligned}\downarrow_{\text{``''}}\ 32 &\twoheadrightarrow \mathit{foldr}\ (:)\ \text{``''}\ 32\\ &\twoheadrightarrow \text{`}3' : \text{`}2' : \mathit{foldr}\ (:)\ \text{``''}\ (\mathit{nil}\ 0)\\ &\twoheadrightarrow \text{`}3' : \text{`}2' : \text{``''}\\ &\twoheadrightarrow \text{``}\mathit{32}\text{''}\end{aligned}$$

With extra knowledge about F and Y we can define some shortcuts.

$$\mathit{foldl}\ (\wedge)\ \bot = \mathbf{K}\ \bot \qquad \mathit{foldl}\ (\vee)\ \top = \mathbf{K}\ \top \qquad \mathit{foldl}\ (\cdot)\ 0 = 0$$

Composition of *foldl* and *map*

$$\mathbf{B}\ (\mathit{foldl}\ F\ Y)\ (\mathit{map}\ G) = \mathit{foldl}\ (\mathbf{C}\ (\mathbf{B}\ \mathbf{B}\ F)\ G)\ Y$$

MapAcc *MapAcc*, a combination of *map* and *foldl*, maps a function over a list L while accumulating information in some environment Y. (The function is from [PL93] where it is called mapAccuml.)

$$\begin{aligned}&\mathit{mapAcc}\ F\ L_0\ Y\ L = \langle Y, L_0\rangle \qquad \Leftarrow \varepsilon?\ L = \top\\ &\mathit{mapAcc}\ F\ L_0\ Y\ L = \langle Y_2, X : L_2\rangle\\ &\qquad\qquad \mathbf{when}\\ &\qquad\qquad\quad \varepsilon?\ L \quad = \bot,\\ &\qquad\qquad\quad \langle Y_1, X\rangle = F\ Y\ (\mathit{hd}\ L),\\ &\qquad\qquad\quad \langle Y_2, L_2\rangle = \mathit{mapAcc}\ F\ L_0\ Y_1\ (\mathit{tl}\ L)\end{aligned}$$

5.3 Concatenation

Imports Map-and-Fold$^{(5.2)}$

Exports
 Context-free Functions
  "++" → Right-Infix-Operator {**type:** $\alpha^* \times \alpha^* \to \alpha^*$}
  concat → Combinator {**type:** $(\alpha^*)^* \to \alpha^*$}
  flatMap → Combinator {**type:** $(\alpha \to \beta^*) \to \alpha^* \to \beta^*$}
Hiddens
 Variables
  [*FGHLPTXYZ*] [*0-9'*]* → Term
  [*XYZ*] "*" → {Term "," }*
  [*XYZ*] "+" → {Term "," }+

Equations

Concatenation (++) concatenates two lists. For []-lists the definition of ++ is

$$[X*] + \!\!+ [Y*] = [X*, Y*]$$

For lists (++) the definition is

$$L_1 + \!\!+ L_2 = \mathit{foldr}\ (:)\ L_2\ L_1$$

Concat concatenates the lists in a list of lists. *Foldl* is specified below. It accumulates the elements in a list with its first functional argument and second initializing argument.

$$\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ []$$

Example *concat* $[[1,2],[3,4],[5,6]]$ reduces to

$$(((\,[\,] +\!\!+ [1,2]) +\!\!+ [3,4]) +\!\!+ [5,6]) \twoheadrightarrow [1,2,3,4,5,6]$$

FlatMap A combination of *concat* and *map* (see [Pey87]). *FlatMap* applies a list valued function F to each element of a list L concatenating the results.

$$\begin{aligned} &\mathit{flatMap}\ F\ L = L && \Leftarrow \varepsilon?\ L = \top \\ &\mathit{flatMap}\ F\ L = F\ (\mathit{hd}\ L) +\!\!+ \mathit{flatMap}\ F\ (\mathit{tl}\ L) && \Leftarrow \varepsilon?\ L = \bot \end{aligned}$$

5.4 Sublists

Imports Lists-and-Tuples$^{(4.6)}$

Exports
 Context-free Functions
 take → Combinator {**type:** $\mathbb{N} \to \alpha^* \to \alpha^*$}
 drop → Combinator {**type:** $\mathbb{N} \to \alpha^* \to \alpha^*$}
 take-while → Combinator {**type:** $(\alpha \to \mathbb{B}) \to \alpha^* \to \alpha^*$}
 drop-while → Combinator {**type:** $(\alpha \to \mathbb{B}) \to \alpha^* \to \alpha^*$}

 filter → Combinator {**type:** $(\alpha \to \mathbb{B}) \to \alpha^* \to \alpha^*$}
 split → Combinator {**type:** $(\alpha \to \mathbb{B}) \to \alpha^* \to \alpha^* \times \alpha^*$}
 split-while → Combinator {**type:** $(\alpha \to \mathbb{B}) \to \alpha^* \to \alpha^* \times \alpha^*$}

Hiddens
 Variables
 [*FGHLPTXYZ*] [*0-9′*]* → Term
 [*XYZ*] "*" → {Term "," }*
 [*XYZ*] "+" → {Term "," }+

Equations

Take and Drop *Take* and *drop* the first n items from a list.

$$\begin{aligned} &\mathit{take}\ 0\ L = \mathit{nil}\ L \\ &\mathit{take}\ n\ L = L && \Leftarrow \varepsilon?\ L = \top \\ &\mathit{take}\ n\ L = \mathit{hd}\ L : \mathit{take}\ (n-1)\ (\mathit{tl}\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ n > 0 = \top \end{aligned}$$

$$\begin{aligned} &\mathit{drop}\ 0\ L = L \\ &\mathit{drop}\ n\ L = L && \Leftarrow \varepsilon?\ L = \top \\ &\mathit{drop}\ n\ L = \mathit{drop}\ (n-1)\ (\mathit{tl}\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ n > 0 = \top \end{aligned}$$

Composition of take and drop ([BW88] p. 56).

$$\mathbf{B}\ (\mathit{drop}\ i)\ (\mathit{drop}\ j) = \mathit{drop}\ (i+j)$$

Take-while and Drop-while *Take-while* and *drop-while* take, respectively drop, items from a list as long as the items satisfy some predicate P.

$$\begin{aligned}
&\textit{take-while}\ P\ L = L && \Leftarrow \varepsilon?\ L = \top\\
&\textit{take-while}\ P\ L = \textit{hd}\ L : \textit{take-while}\ P\ (\textit{tl}\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \top\\
&\textit{take-while}\ P\ L = \textit{nil}\ L && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \bot
\end{aligned}$$

$$\begin{aligned}
&\textit{drop-while}\ P\ L = L && \Leftarrow \varepsilon?\ L = \top\\
&\textit{drop-while}\ P\ L = \textit{drop-while}\ P\ (\textit{tl}\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \top\\
&\textit{drop-while}\ P\ L = L && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \bot
\end{aligned}$$

Filter takes a predicate and a list and yields the list of all elements that satisfy the predicate.

$$\begin{aligned}
&\textit{filter}\ P\ L = L && \Leftarrow \varepsilon?\ L = \top\\
&\textit{filter}\ P\ L = \textbf{\textit{if}}\ (P\ X)\ (X :)\ \mathbf{I}\ (\textit{filter}\ P\ (\textit{tl}\ L)) && \Leftarrow \varepsilon?\ L = \bot,\ \ X = \textit{hd}\ L
\end{aligned}$$

Sometimes we do not only want the elements that satisfy the predicate but also the elements that do not. That is *split*ting a list in a pair of lists; one that contains the $\top$ items with respect to the predicate, and one containing the $\bot$ items.

$$\begin{aligned}
&\textit{split}\ P\ L = \langle L,\ L\rangle && \Leftarrow \varepsilon?\ L = \top\\
&\textit{split}\ P\ L = \langle \textit{hd}\ L : L_1,\ L_2\rangle && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \top,\ \ \langle L_1,\ L_2\rangle = \textit{split}\ P\ (\textit{tl}\ L)\\
&\textit{split}\ P\ L = \langle L_1,\ \textit{hd}\ L : L_2\rangle && \Leftarrow \varepsilon?\ L = \bot,\ \ P\ (\textit{hd}\ L) = \bot,\ \ \langle L_1,\ L_2\rangle = \textit{split}\ P\ (\textit{tl}\ L)
\end{aligned}$$

Split-while splits a list L in the pair (L_1, L_2), such that for all elements of L_1 P holds, and that L_2 is either empty or P does not hold for the head of L_2.

$$\textit{split-while}\ P\ L = \langle \textit{take-while}\ P\ L,\ \textit{drop-while}\ P\ L\rangle$$

5.5 Sets

Imports Map-and-Fold(5.2) Sublists(5.4)

Exports
 Context-free Functions
  "$\bigwedge$" $\rightarrow$ Combinator {**type:** $\mathbb{B}^* \rightarrow \mathbb{B}$}
  "$\bigvee$" $\rightarrow$ Combinator {**type:** $\mathbb{B}^* \rightarrow \mathbb{B}$}
  "$\forall$" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \mathbb{B}) \rightarrow \alpha^* \rightarrow \mathbb{B}$}
  "$\exists$" $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \mathbb{B}) \rightarrow \alpha^* \rightarrow \mathbb{B}$}

  "$\in$" $\rightarrow$ Infix-Operator {**type:** $\alpha \times \alpha^* \rightarrow \mathbb{B}$}
  "$\subseteq$" $\rightarrow$ Infix-Operator {**type:** $\alpha^* \times \alpha^* \rightarrow \mathbb{B}$}
  "$\cup$" $\rightarrow$ Infix-Operator {**type:** $\alpha^* \times \alpha^* \rightarrow \alpha^*$}
  "$\cap$" $\rightarrow$ Infix-Operator {**type:** $\alpha^* \times \alpha^* \rightarrow \alpha^*$}
  "$\backslash$" $\rightarrow$ Infix-Operator {**type:** $\alpha^* \times \alpha^* \rightarrow \alpha^*$}
  "*card*" $\rightarrow$ Combinator {**type:** $\alpha^* \rightarrow \mathbb{N}$}

Hiddens
 Context-free Functions
  add $\rightarrow$ Combinator
 Variables
  [*ABPX*] [*0-9'*]∗ $\rightarrow$ Term

Equations

The generalisation of $\wedge$ and $\vee$ for lists of booleans.

$$\bigwedge = \mathit{foldl}\ (\wedge)\ \top \qquad \bigvee = \mathit{foldl}\ (\vee)\ \bot$$

The predicates $\forall$ and $\exists$ check, respectively, if all or some items in a list satisfy a unary predicate P.

$$\forall\ P\ A = \mathit{foldr}\ ((\wedge) \circ P)\ \top\ A \qquad \exists\ P\ A = \mathit{foldr}\ ((\vee) \circ P)\ \bot\ A$$

Negation of $\forall$ and $\exists$

$$/\ (\forall\ P) = \exists\ (/\ P) \qquad /\ (\exists\ P) = \forall\ (/\ P)$$

Member X is a member $(\in)$ of A when A has an element equal (in the sense of $(\equiv)$) to X.

$$X \in A = \exists\ (X \equiv)\ A$$

Sublist A list A is a sublist of a list B if all elements of A are members of B.

$$A \subseteq B = \forall\ (\in B)\ A$$

Union $A \cup B$ is the list of all elements of A and B (in that order) with duplicates deleted.

$$A \cup B = \mathit{foldr}\ \mathit{add}\ (\mathit{foldr}\ \mathit{add}\ (\mathit{nil}\ B)\ B)\ A$$

where *add* is defined as

$$\mathit{add}\ X\ A = \boldsymbol{\mathit{if}}^c\ \langle X \in A, \mathbf{I}, (X :)\rangle\ A$$

Intersection

$$A \cap B = \mathit{foldr}\ \mathit{add}\ (\mathit{nil}\ B)\ (\mathit{filter}\ (\in B)\ A)$$

Set-Minus

$$A \setminus B = \mathit{filter}\ (/\ (\in B))\ A$$

Cardinality

$$\begin{aligned} \mathit{card}\ A &= 0 && \Leftarrow \varepsilon?\ A = \top \\ \mathit{card}\ A &= 1 + \mathit{card}\ (A \setminus \mathit{hd}\ A : \mathit{nil}\ A) && \Leftarrow \varepsilon?\ A = \bot \end{aligned}$$

5.6 List Predicates

Imports Map-and-Fold$^{(5.2)}$ Sublists$^{(5.4)}$

Exports

Context-free Functions

compare $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \mathbb{B}) \rightarrow \alpha^* \rightarrow \beta^* \rightarrow \mathbb{B}$}

“#” $\rightarrow$ Combinator {**type:** $\alpha^* \rightarrow \mathbb{N}$}
count $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \mathbb{B}) \rightarrow \alpha^* \rightarrow \mathbb{N}$}
reverse $\rightarrow$ Combinator {**type:** $\alpha^* \rightarrow \alpha^*$}
apply-list $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \alpha \rightarrow \beta^*$}

Hiddens

Variables

[*FGHLPTXYZ*] [*0-9′*]∗ $\rightarrow$ Term
[*XYZ*] “∗” $\rightarrow$ {Term “,” }∗
[*XYZ*] “+” $\rightarrow$ {Term “,” }+

Equations

Compare compares, given a predicate P and two lists L_1 and L_2, the heads of the lists with P and recursively compares the tails of the lists. Comparing lists of unequal length yields $\bot$.

$$compare\ P\ L_1\ L_2 = \top \Leftarrow \varepsilon?\ L_1 = \top,\ \ \varepsilon?\ L_2 = \top$$
$$compare\ P\ L_1\ L_2 = \bot \Leftarrow \varepsilon?\ L_1 = \bot,\ \ \varepsilon?\ L_2 = \top$$
$$compare\ P\ L_1\ L_2 = \bot \Leftarrow \varepsilon?\ L_1 = \top,\ \ \varepsilon?\ L_2 = \bot$$

$$compare\ P\ L_1\ L_2 = P\ (hd\ L_1)\ (hd\ L_2) \wedge compare\ P\ (tl\ L_1)\ (tl\ L_2)$$

when

$$\varepsilon?\ L_1 = \bot,\ \ \varepsilon?\ L_2 = \bot$$

Equality ($\equiv$) **of Lists and Tuples** With *compare* we can extend the definition of the equality predicate ($\equiv$) we defined in Module 4.4.

$$[X*] \equiv [Y*]\ = compare\ (\equiv)\ [X*]\ [Y*]$$
$$\langle X*\rangle \equiv \langle Y*\rangle = compare\ (\equiv)\ \langle X*\rangle\ \langle Y*\rangle$$

List Length # gives the length of a list.

$$\#\, L = 0 \Leftarrow \varepsilon?\ L = \top \qquad \#\, L = 1 + \#\ (tl\ L) \Leftarrow \varepsilon?\ L = \bot$$

Count counts the number of items in a list that satisfy some predicate P. It is implemented by first filtering the ‘right’ items and taking the length of the resulting list.

$$count\ P = \# \circ filter\ P$$

Reverse reverses the order of the items in a list.

$$reverse\ L = nil\ L \qquad \Leftarrow \varepsilon?\ L = \top$$
$$reverse\ L = last\ L : reverse\ (init\ L) \Leftarrow \varepsilon?\ L = \bot$$

Apply-list applies each function in a list of functions to an argument yielding the list of the results (note the use of $\mathbf{A}_2$).

$$apply\text{-}list\ L\ X = map\ (\mathbf{A}_2\ X)\ L$$

5.7 Sorted Lists

Imports Map-and-Fold[5.2]

Exports
Context-free Functions
"$\sum$" $\to$ Combinator {**type:** $\mathbb{N}^* \to \mathbb{N}$}
"$\prod$" $\to$ Combinator {**type:** $\mathbb{N}^* \to \mathbb{N}$}
sort $\to$ Combinator {**type:** $(\alpha \to \alpha \to \mathbb{B}) \to \alpha^* \to \alpha^*$}
insert $\to$ Combinator {**type:** $(\alpha \to \alpha \to \mathbb{B}) \to \alpha \to \alpha^* \to \alpha^*$}
unique $\to$ Combinator {**type:** $\alpha^* \to \alpha^*$}
".." $\to$ Infix-Operator {**type:** $\mathbb{N} \times \mathbb{N} \to \mathbb{N}^*$}

Hiddens
Variables
[*FGHLPTXYZ*] [*0-9'*]* $\to$ Term
[*XYZ*] "*" $\to$ {Term "," }*
[*XYZ*] "+" $\to$ {Term "," }+

Equations

Sum and Product The sum and product of a list of integers are computed respectively, by combinators $\sum$ and $\prod$.

$$\sum = foldl\ (+)\ 0 \qquad \prod = foldl\ (\cdot)\ 1$$

Sorting Given some linear order P *sort* sorts a list with the insertion sort algorithm in ascending P-order.

$$sort\ P\ L = foldr\ (insert\ P)\ (nil\ L)\ L$$

$$\begin{aligned}
&insert\ P\ X\ L = X : L && \Leftarrow \varepsilon?\ L = \top\\
&insert\ P\ X\ L = X : L && \Leftarrow \varepsilon?\ L = \bot,\ \ Y = hd\ L,\ \ P\ X\ Y = \top\\
&insert\ P\ X\ L = Y : insert\ P\ X\ (tl\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ Y = hd\ L,\ \ P\ X\ Y = \bot
\end{aligned}$$

Unique reduces neighbouring strings of equal items in a list to one item.

$$\begin{aligned}
&unique\ L = L && \Leftarrow \varepsilon?\ L = \top\\
&unique\ L = L && \Leftarrow \varepsilon?\ L = \bot,\ \ \varepsilon?\ (tl\ L) = \top\\
&unique\ L = unique\ (tl\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ \varepsilon?\ (tl\ L) = \bot,\ \ hd\ L = hd\ (tl\ L)\\
&unique\ L = hd\ L : unique\ (tl\ L) && \Leftarrow \varepsilon?\ L = \bot,\ \ \varepsilon?\ (tl\ L) = \bot,\ \ hd\ L \neq hd\ (tl\ L)
\end{aligned}$$

With *sort* and *unique* we can sort a list of integers and then remove all duplicates by $(unique \circ sort(\leq))$.

Interval The list of integers from some integer n_1 up- or downto some integer n_2, i.e. the interval $[n_1, \ldots, n_2]$ is constructed by the infix-operator (..).

$$\begin{aligned}
&n_1\ ..\ n_2 = [] && \Leftarrow n_1 > n_2 = \top\\
&n_1\ ..\ n_2 = n_1 : n_1 + 1\ ..\ n_2 && \Leftarrow n_1 \leq n_2 = \top
\end{aligned}$$

5.8 Zip

Imports Sets[(5.5)]

Exports

Context-free Functions

zip $\rightarrow$ Combinator {**type:** $\alpha^* \rightarrow \beta^* \rightarrow (\alpha \times \beta)^*$}
unzip $\rightarrow$ Combinator {**type:** $(\alpha \times \beta)^* \rightarrow (\alpha^* \times \beta^*)$}
zip-with $\rightarrow$ Combinator {**type:** $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha^* \rightarrow \beta^* \rightarrow \gamma^*$}

Hiddens

Variables

[*FGHLPTXYZ*] [*0-9'*]* $\rightarrow$ Term
[*XYZ*] "*" $\rightarrow$ {Term "," }*
[*XYZ*] "+" $\rightarrow$ {Term "," }+

Equations

Zip turns a tuple or list of lists into a list of the tuples of the heads of the lists.

$$zip\ L = [] \Leftarrow \varepsilon?\ L \vee \exists\ \varepsilon?\ L = \top$$

$$zip\ L = map\ hd\ L : zip\ (map\ tl\ L) \Leftarrow \neg\ (\varepsilon?\ L) \wedge /\ (\exists\ \varepsilon?)\ L = \top$$

Unzip turns a list of pairs into a pair of lists.

$$unzip\ L = \langle L,\ L\rangle \Leftarrow \varepsilon?\ L = \top$$

$$unzip\ L = \langle X : L_1,\ Y : L_2\rangle \Leftarrow \langle X,\ Y\rangle = hd\ L,\ \ \langle L_1,\ L_2\rangle = unzip\ (tl\ L)$$

Zip-with is a generalization of *zip*; instead of constructing pairs, a construction function F is applied to the heads of the lists.

$$zip\text{-}with\ F\ L_1\ L_2 = [] \Leftarrow \varepsilon?\ L_1 = \top$$

$$zip\text{-}with\ F\ L_1\ L_2 = [] \Leftarrow \varepsilon?\ L_2 = \top$$

$$zip\text{-}with\ F\ L_1\ L_2 = F\ (hd\ L_1)\ (hd\ L_2) : zip\text{-}with\ F\ (tl\ L_1)\ (tl\ L_2) \Leftarrow \varepsilon?\ L_1 = \bot,\ \ \varepsilon?\ L_2 = \bot$$

5.9 List Comprehensions

Imports Functional-Abstraction[(4.7)] Concatenation[(5.3)] Sublists[(5.4)]

Miranda has a list comprehension construct to express unnamed operations on list. We specify a subset of this construct. [We do not consider pattern matching λ-abstraction; expressions like $(\lambda\langle\ \_x, \_y\rangle.(\_x + \_y))$ that only applies to pairs.]

Exports

Sorts Qualifier

Context-free Functions

"[" Term "|" {Qualifier ";" }* "]" $\rightarrow$ Term {**type:** type(Term$_1$)*}
Variable "$\leftarrow$" Term $\rightarrow$ Qualifier
if Term $\rightarrow$ Qualifier

Remark The typing of list comprehensions can be defined stricter than is done here, when the types of the qualifiers are taken into account. Some mechanism to transport the types of bound variables is needed here.

Hiddens

Variables

"$\mathcal{Q}$" [*0-9'*]$*$	$\rightarrow$	Qualifier
"$\vec{\mathcal{Q}}$" [*0-9'*]$*$	$\rightarrow$	{Qualifier ";" }$*$
[*BLMPXYZ*] [*0-9'*]$*$	$\rightarrow$	Term
[*XYZ*] "$*$"	$\rightarrow$	{Term "," }$*$

Remark $\mathcal{V} \leftarrow L$ is called a generator, $if P$ a predicate.

Equations

Semantics of List Comprehensions as given in [Pey87] (p.132).

$$[X \mid\,] = [X]$$

$$[X \mid \mathcal{V} \leftarrow L;\, \vec{\mathcal{Q}}] = \mathit{flatMap}\ (\lambda\ \mathcal{V}\ .\ [X \mid \vec{\mathcal{Q}}])\ L$$

$$[X \mid \mathit{if}\, B;\, \vec{\mathcal{Q}}] = \boldsymbol{\mathit{if}}\, B\, [X \mid \vec{\mathcal{Q}}]\ []$$

Example The Cartesian product $L_1 \times L_2$ of lists L_1 and L_2 can be defined with a list comprehension as

$$L_1 \times L_2 = [\langle \_x, \_y\rangle | \_x \leftarrow L_1; \_y \leftarrow L_2] \qquad \bigstar$$

For instance, if $L_1 = [1, 2]$ and $L_2 = [4, 5, 6]$, then

$$\begin{aligned} L_1 \times L_2 &= [\langle \_x, \_y\rangle | \_x \leftarrow [1, 2]; \_y \leftarrow [4, 5, 6]] \\ &= [\langle 1, 4\rangle, \langle 1, 5\rangle, \langle 1, 6\rangle, \langle 2, 4\rangle, \langle 2, 5\rangle, \langle 2, 6\rangle] \end{aligned}$$

Example (Multiplication tables) The following expression specifies a multiplication table:

$$[\langle \_y, [\langle \_x, \_x \cdot \_y\rangle | \_x \leftarrow 1..10]\rangle | \_y \leftarrow 1..10]$$

for $\_y = 1 \ldots 10$ a pair $\langle n, T\rangle$ is formed, where T is a table containing for each $\_x \in \{1, \ldots, 10\}$ a pair $\langle \_x, \_x \cdot \_y\rangle$.

5.10 Tables

Imports Lists-and-Tuples$^{(4.6)}$

A table is a list of table entries, which have a key accesible by combinator *key* and a value accesible by combinator *value*. Three operations are provided. Lookup (?) checks a table for occurrence of some key. Value (.) looks up the value atached to some key in the table. Remove (*rm*) removes all entries with some key from the table.

We define the types Entry(α, β) of entries and Table(α, β) of tables mapping α's to β's.

Exports

Context-free Functions

key	$\rightarrow$ Combinator	{**type:** Entry$(\alpha, \beta) \rightarrow \alpha$}
value	$\rightarrow$ Combinator	{**type:** Entry$(\alpha, \beta) \rightarrow \beta$}
"\?"	$\rightarrow$ Infix-Operator	{**type:** Table$(\alpha, \beta) \times \alpha \rightarrow \mathbb{B}$}
"."	$\rightarrow$ Infix-Operator	{**type:** Table$(\alpha, \beta) \times \alpha \rightarrow \beta$}

“rm” $\rightarrow$ Combinator $\{$**type:** $\text{Table}(\alpha, \beta) \rightarrow \alpha \rightarrow \text{Table}(\alpha, \beta)\}$

Hiddens

Variables

$[KLTXY]\ [']*$ $\rightarrow$ Term
$[XYLT]\ [*]\ [0\text{-}9']*$ $\rightarrow$ $\{$Term “,” $\}*$
$[XYLT]\ [+]\ [0\text{-}9']*$ $\rightarrow$ $\{$Term “,” $\}+$

Equations

Key and Value When $\langle\rangle$-tuples are used as table entries, the key is the first element of the tuple and the value, either the second element in case of a pair or the rest of the tuple in case of an n-ary tuple.

$$\mathit{key}\ \langle K, X+\rangle = K \qquad \mathit{value}\ \langle K, X\rangle = X \qquad \mathit{value}\ \langle K, X, X+\rangle = \langle X, X+\rangle$$

Entry Defined?

$$\begin{aligned}
T\ ?\ K &= \bot && \Leftarrow \varepsilon?\ T = \top \\
T\ ?\ K &= \top && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) = K \\
T\ ?\ K &= \mathit{tl}\ T\ ?\ K && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) \neq K
\end{aligned}$$

Lookup Value

$$\begin{aligned}
T\ .\ K &= \mathit{value}\ (\mathit{hd}\ T) && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) = K \\
T\ .\ K &= \mathit{tl}\ T\ .\ K && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) \neq K
\end{aligned}$$

Remove

$$\begin{aligned}
\mathit{rm}\ K\ T &= T && \Leftarrow \varepsilon?\ T = \top \\
\mathit{rm}\ K\ T &= \mathit{rm}\ K\ (\mathit{tl}\ T) && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) = K \\
\mathit{rm}\ K\ T &= \mathit{hd}\ T : \mathit{rm}\ K\ (\mathit{tl}\ T) && \Leftarrow \varepsilon?\ T = \bot,\ \ \mathit{key}\ (\mathit{hd}\ T) \neq K
\end{aligned}$$

Chapter 6

Tree Operations

Trees are a commonly used datatype in compilers (parse trees and abstract syntax trees) and other applications. Here, we define a couple of functions to manipulate terms over any sort as trees.

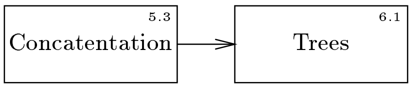


Figure 6.1: Import graph for Chapter 6; Tree Operations

6.1 Trees

Imports Concatenation$^{(5.3)}$

We define functions, similar to the primitive list operations, for trees. Each node of a tree is constructed from a label and a list of sons. By defining the operations *tree*, *label* and *sons* for an arbitrary sort we can apply tree traversal functions to terms over that sort.

Exports

Context-free Functions

tree → Combinator {**type:** $\alpha \to \text{Tree}(\alpha)^* \to \text{Tree}(\alpha)$}
label → Combinator {**type:** $\text{Tree}(\alpha) \to \alpha$}
sons → Combinator {**type:** $\text{Tree}(\alpha) \to \text{Tree}(\alpha)^*$}

subtrees → Combinator {**type:** $\text{Tree}(\alpha) \to \text{Tree}(\alpha)^*$}
labels → Combinator {**type:** $\text{Tree}(\alpha) \to \alpha^*$}

preorder → Combinator {**type:** $(\text{Tree}(\alpha) \to \text{Tree}(\alpha)) \to \text{Tree}(\alpha) \to \text{Tree}(\alpha)$}
postorder → Combinator {**type:** $(\text{Tree}(\alpha) \to \text{Tree}(\alpha)) \to \text{Tree}(\alpha) \to \text{Tree}(\alpha)$}

poAcc → Combinator {**type:** $(\gamma \to \delta \to \alpha^* \to \gamma \times \alpha) \to \alpha^* \to \gamma \to \text{Tree}(\delta) \to \gamma \times \alpha$}

Variables

"Tree" [*0-9′*]∗ → Term
"Tree*" [*0-9′*]∗ → {Term "," }∗
"*Label*" [*0-9′*]∗ → Term
"*Sons*" [*0-9′*]∗ → Term

Hiddens

Variables

[*FGLY*] [*0-9′*]∗ → Term
[*X*] [∗] [*0-9′*]∗ → {Term "," }∗

Equations

Requirements A tree is constructed by combinator *tree*. It turns a label *Label* and a list of trees (the sons) *Sons* into a tree. *Label* and *sons* give the label and the list of sons of a tree.

$$\textit{label}\ (\textit{tree}\ \textit{Label}\ \textit{Sons}) = \textit{Label} \qquad \textit{sons}\ (\textit{tree}\ \textit{Label}\ \textit{Sons}) = \textit{Sons}$$

Subtrees The list of subtrees of a tree.

$$\textit{subtrees}\ \mathrm{Tree} = \mathrm{Tree} : \textit{flatMap}\ \textit{subtrees}\ (\textit{sons}\ \mathrm{Tree})$$

Labels The list of labels of a tree.

$$\textit{labels}\ \mathrm{Tree} = \textit{map}\ \textit{label}\ (\textit{subtrees}\ \mathrm{Tree})$$

Preorder Traversal Process tree before all its subtrees. (See also [SW85].) This means that the function F should yield a tree when applied to a tree.

$$\frac{\mathrm{Tree}' = F\ \mathrm{Tree}}{\textit{preorder}\ F\ \mathrm{Tree} = \textit{tree}\ (\textit{label}\ \mathrm{Tree}')\ (\textit{map}\ (\textit{preorder}\ F)\ (\textit{sons}\ \mathrm{Tree}'))}$$

Postorder Traversal Process the tree after processing all its subtrees.

$$\textit{postorder}\ F\ \mathrm{Tree} = F\ (\textit{tree}\ (\textit{label}\ \mathrm{Tree})\ (\textit{map}\ (\textit{postorder}\ F)\ (\textit{sons}\ \mathrm{Tree})))$$

Postorder Accumulate *poAcc* does a postorder traversal over a tree while accumulating information in an environment.

$$\frac{\langle Y', L'\rangle = \textit{mapAcc}\ (\textit{poAcc}\ F\ L)\ L\ Y\ (\textit{sons}\ \mathrm{Tree})}{\textit{poAcc}\ F\ L\ Y\ \mathrm{Tree} = F\ Y'\ (\textit{label}\ \mathrm{Tree})\ L'}$$

Note that *preorder* and *postorder* are analogous to *map* and that *poAcc* is analogous to *mapAcc*. There should also be anologues of *foldr* and *foldl* for trees. I did not yet find applications that suggest what these functions should look like.

Chapter 7

Evaluation

We have extended ASF+SDF with an untyped functional programming language in the style of Miranda [BW88]. The approach has clear advantages but also some drawbacks.

7.1 Assets

The goal was to develop a method that enables the definition of generic reusable functions. The approach to this problem was the design of a simple untyped applicative language in the style of Miranda [BW88]. The syntax of this language only defines terms; ASF+SDF equations are used to define the semantics of terms. Arbitrary syntactic constructs can be added to the language as constants.

Functional Programming + Algebraic Specification The resulting *combinatory* style of algebraic specification combines the advantages of both functional programming and algebraic specification.

From functional programming the treatment of functions as first-class citizens is inherited. Functions or *combinators* can be manipulated like first-order objects. In particular, functions can be arguments of other functions.

In algebraic specification arbitrary complex patterns can be used in the left-hand side of equations. A generic equation like

$$F\ (\mathbf{if}\ B\ X\ Y) = \mathbf{if}\ B\ (F\ X)\ (F\ Y)$$

where F is a (higher-order) variable, can not be written in a Miranda program. Another difference is that the equations defining a function may be spread over more than one module.

A feature that is neither encountered in first-order algebraic specification nor in functional programming is that extensional shortcuts for special cases can be defined. For instance, we defined the function *foldl* by the equations

$$\begin{aligned} &\mathit{foldl}\ F\ Y\ L = L \Leftarrow \varepsilon?\ L = \top \\ &\mathit{foldl}\ F\ Y\ L = \underline{\mathit{foldl}\ F\ (F\ Y\ (\mathit{hd}\ L))}\ (\mathit{tl}\ L) \Leftarrow \varepsilon?\ L = \bot \end{aligned} \tag{7.1}$$

The term $(\mathit{foldl}\ (\wedge)\ \top)$ then represents the function that yields the conjunction of all elements in a list of Boolean values. For instance,

$$\begin{aligned} \mathit{foldl}\ (\wedge)\ \top\ [\bot,\top,\top] &\twoheadrightarrow \mathit{foldl}\ (\wedge)\ \bot\ [\top,\top] \\ &\twoheadrightarrow \mathit{foldl}\ (\wedge)\ \bot\ [\top] \twoheadrightarrow \mathit{foldl}\ (\wedge)\ \bot\ [] \twoheadrightarrow \bot \end{aligned}$$

However, we know that the function will yield $\perp$ as soon as the value $\perp$ is encountered. In order to break off the traversal of the rest of the list, the equation

$$foldl\,(\wedge)\ \perp = \mathbf{K}\ \perp \tag{7.2}$$

is added. We then have the reduction

$$foldl\,(\wedge)\ \top\ [\perp, \top, \top] \twoheadrightarrow foldl\,(\wedge)\ \perp\ [\top, \top] \twoheadrightarrow \mathbf{K}\ \perp\ [\top, \top] \rightarrow \perp$$

This works due to innermost reduction. The underlined term in the right-hand side of equation (7.1) is reduced before the entire rhs is reduced. That term is reduced by (7.2) to $\perp$ if F is $(\wedge)$ and $(F\ Y\ (hd\ L))$ reduces to $\perp$. The recursive call to *foldl* is thereby inhibited. This mechanism can be used to optimize functions without having to spell out the recursion scheme (which we wanted to avoid).

Outermost Graph Reduction vs. Innermost Tree Reduction A difference with functional programming is the implementation. Languages like Miranda have outermost graph reduction as execution model while algebraic specification (ASF) has innermost tree reduction as execution model.

The representation of terms as graphs in Miranda entails that terms may contain cycles. Circular terms can be used to represent infinite datastructures. Although sharing is accounted for in the ASF+SDF implementation by allowing multiple parents of a tree (leading to directed acyclic graphs) cycles in terms are not possible.

Outermost reduction will terminate in more cases than innermost reduction. A term with a subterm that never reduces to a normal form will lead in ASF+SDF to a non-terminating reduction, while in Miranda the reduction may terminate if the subterm is not needed.

Library of Reusable Combinators The combinatory language extended with λ abstraction and list comprehensions and with an extensive library of reusable combinators leads to an expressive extension to the normal, first-order use of ASF+SDF.

The strength of this method in terms of development effort is illustrated by the fact that a generic unification function for trees, parameterized with a notion of variables, was written in about one hour. Subsequently an interpreter for a simple logic programming language, with the essential features of Prolog, was written in another hour.

Despite these advantages, the approach has some shortcomings.

7.2 Problems

The method developed here has several drawbacks. The most serious of these is the fact that the combinatory language is untyped. A problem that is associated with any higher-order deduction system is extensional equality. Finally, combinatory specifications behave less efficiently as a term rewrite system than first-order specifications.

7.2.1 Static Correctness

As remarked in Chapter 3 the distinction between terms of different sorts is lost. The function $(+)$ and the constant 0 are both members of sort Term. Furthermore, we can construct syntactically correct terms that are meaningless from our point of view. For instance, $(0\ (+))$, the application of 0 (a constant) to $(+)$ (a function), is syntactically valid.

We introduced the mechanism of type annotations to indicate our intention with respect to the types of terms. However, this mechanism only serves documentation purposes and has no formal status.

Of course we can (and do) restrict ourselves to meaningful terms. In this setting there is, however, no formal way to say what meaningful and meaningless terms are.

A related problem is the fact that combinators are not functions in the usual sense. Although we think of *map* as a function with arity 2, it is, in this specification, a special kind of constant; a function with arity 0.

Below we will discuss some possibilities to repair this problem.

7.2.2 Intensionality of Equality

A problem that can not be repaired easily is the non-extensional equality of functional objects. This leads to the problem that the inequality condition of ASF+SDF has to be used with care.

7.2.3 Efficiency

The behaviour of a combinatory specifications as term rewrite systems is less efficient than that of first-order specifications, both in time and space.

Time In general, equations in the combinatory style can be written that result in the same number of reduction steps as in a first-order style. However, the full exploitation of the higher-order features leads to extra reduction steps. For instance, many list operations make use of the primitive list operations. A list is analyzed with $\varepsilon?$, head and tail are extracted by *hd* and *tl* and new lists are constructed with *nil* and (:). These operations cost reduction steps that are not done in a first-order style where analysis, extraction are done by direct matching of the list and construction is done directly. As an illustration consider the following reductions. In brackets, we show the reduction steps of conditions. The first reduction shows the mapping of the function (+1) over the list [2].

$$\begin{array}{rcll} \mathit{map}\ (+1)\ [2] & \to & (+1)\ 2 : \mathit{map}\ (+1)\ [] & (\varepsilon?\ [2] \to \bot) \\ & \to & (2+1) : \mathit{map}\ (+1)\ [] & \\ & \to & [\![2+1]\!] : \mathit{map}\ (+1)\ [] & \\ & \twoheadrightarrow & 3 : \mathit{map}\ (+1)\ [] & \\ & \to & 3 : [] & (\varepsilon?\ [] \to \top) \\ & \to & [3] & \end{array}$$

The equivalent first order function *map-plus-one* has the following reduction when applied to the same list.

$$\mathit{map\text{-}plus\text{-}one}\ [2] \to [3] \quad (2+1 \twoheadrightarrow 3, \mathit{map\text{-}plus\text{-}one}\ [] \to [])$$

The use of λ expressions is discouraged while constant expressions, without variables that are instantiated, are always reduced to the same term. For instance, the expression

$$\lambda\_x\_y.\langle\_x, \_y\rangle$$

has the normal form

$$\mathbf{C}\ (\mathbf{B}\ \mathbf{B}\ \mathbf{P}_n)\ (\mathbf{C}\ \mathbf{P}_n\ \langle\ \rangle)$$

but every time a right-hand side with this expression is instantiated that normal form is computed.

Finally, we note that all rules over applications are definitions of the function

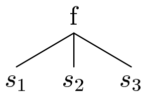
$$\text{Term Term} \to \text{Term}$$

When reducing an application all equations defining this function have to be considered. This is in contrast with our intuition that an equation with left-hand side ($map\ F\ L$) defines the function map.

Space Combinatory terms occupy more memory than the associated first-order terms. For instance, consider the first-order function

$$f(\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3) \to \mathrm{S},$$

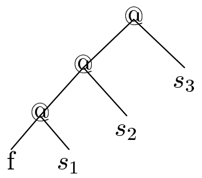
Let s_i be terms of sort S_i, then the term $f(s_1, s_2, s_3)$ is represented by the tree



We will count each treenode and each pointer as one memory unit. Then, discarding the space occupied by the s_i, this term occupies 4 units. Now, consider the combinator

$$f \to \text{Combinator } \{\textbf{type: } \mathrm{S}_1 \to \mathrm{S}_2 \to \mathrm{S}_3 \to \mathrm{S}\}$$

The term $(f\,s_1\ s_2\ s_3)$ is represented by the tree



where @ denotes the combinatory application operator. This term occupies 4 nodes and 6 pointers.

In general, the application of a combinator to n arguments will need n more nodes and n more pointers than the equivalent first-order term.

However, in our style of specification combinators are not used as constructors of datatypes. For this purpose lists, tuples and infix-operators are used. This means that applications of combinators to enough arguments will not be normal forms. The extra space occupied by applications will generally soon after allocation be freed. This is an extra burden for garbage collection.

7.3 The Future of Higher-Order Algebraic Specification

It is commonly agreed among ASF+SDF users that some kind of higher-order extension to the formalism would make specification in ASF+SDF more attractive. From the algebraic viewpoint it is interesting to be able to define properties of functions as objects. From the software engineering viewpoint, higher-order features can be an answer to the problem of making functions reusable.

In the ASF+SDF community a discussion, about which (higher-order) features to add to the formalism, is going on. Two directions can be distinguished.

The first [Hee92a, Hee92b], is concerned with a matching strategy, called *higher-order matching*, that is more powerful than the first-order matching strategy used in the current implementation. This approach leads to a more declarative style of specification.

The second direction [Kli92, Gan93] concentrates on adding a feature to the system that enables one to extract the function and arguments from a first-order term in a generic way. The functional object thus derived can be used as a higher-order argument to other functions. To use this feature a language similar to the combinatory language defined here is used, with similar typing problems.

Finally we mention [Deu92a], which describes an environment for the manipulation of λ expression. Beta reduction is specified using substitution.

We discuss two possibilities to integrate higher-order features in the framework of ASF+SDF.

Built in Higher-Order Types in ASF+SDF Typesystem The typesystem of ASF+SDF is built on the type constructor $(\prod \mathrm{A}_i) \rightarrow \mathrm{A}$. That is, n-ary functions over sort names. One can imagine an extension of this typesystem with constructors as proposed in Section 3.3 including type variables. It is not yet totally clear what the consequences of such an extension are to the grammars derived from signatures.

User Definable Type Constructors In [Mei91] the algebraic framework for a system with 'user definable' type constructors is described. A specification consists of a meta-specification which defines the algebra of types. The signature of the 'normal' specification uses terms over this algebra as types. This was the idea behind the type annotations in described in Chapter 3. Module Higher-Order-Types specifies a typesystem, that was subsequently used as the typesystem for the specification of combinators. This approach would make the definition of higher-order features much more flexible.

Both possibilities leave a choice to the matching mechanism used.

7.4 Writing Literate Specifications

In the concept of 'literate programming' Knuth [Knu84] included reviewing of literate programs, similar to reviews of works of literature. Although such reviews are usually done by others than the author, it seems appropriate to make some remarks about the experiences with this style of presentation.

There seems to be a friction between the modularity of the specification and the modularity of the text. The text of a module tends to be very local. The global view of the document is sometimes lost. This is related directly by the locality of modules. In writing a module, the aim is to make it as independent as possible from other modules. This aim is contrary to the aim of writing a text, where the connection between the individual parts is important.

The original WEB for Pascal language is directed at the order of the explanation. However, the approach, of deriving the specification from a WEB, is not desirable in our context. It would mean the loss of the interactive features of the ASF+SDF system.

It is my opinion that these problems are not due to the principles of literate programming (although some think so) but due to lack of experience with writing in general and with this kind of writing in particular by the author. Further experiments with this technique are needed to develop a good 'literate specification' style. It remains a debated question whether specifications should be part of theses, reports or articles. In my opinion, exchange of specifications is essential to the development of algebraic specification as a mature software development tool.

Part II

Some Transformations on Term Rewrite Systems

Chapter 8

Introduction to Part II

8.1 Associative Lists

In plain algebraic specification (for instance, in pure ASF [BHK89]) list sorts can be defined by defining two constructors *cons* and *nil*, with types

$$\begin{aligned} nil &: \text{Item-List} \\ cons &: \text{Item} \times \text{Item-List} \rightarrow \text{Item-List} \end{aligned}$$

In ASF+SDF sort Item-List can be defined by

$$\text{"\{" \{Item ","\} * "\}"} \rightarrow \text{Item-List}$$

With this definition the list $cons(i_0, cons(i_1, nil))$, where i_0 and i_1 are Items, can be written as $\{i_0, i_1\}$.

A special kind of variables, so-called *sublist variables*, can be used in terms, denoting an arbitrary sublist. For instance, the term $\{i_0, I*, i_1\}$ denotes all lists which start with i_0 and end with i_1.

These kinds of lists are called *associative lists* because all elements of the list are equally well accessible. For instance, the terms $\{I, I*\}$ and $\{I*, I\}$ denote lists with I as first and last element, respectively. We can denote an arbitrary element of a list by a term like $\{I1*, I, I2*\}$.

List Matching Associative lists require a special matching strategy. In matching a list pattern $\{I1*, i_0, I2*\}$ with a list, sublists of the list have to be assigned to the sublist variables $I1*$ and $I2*$. After assigning a sublist to $I1*$ it may happen that the next element in the list does not match with i_0. In that case another assignment for $I1*$ has to be found, if one exists.

This entails that the matching machine must keep track of which assignments to variables have been made. If a match is found, it can then still be rejected by one of the conditions of the rule. In this case the rewrite machine must backtrack to the last list match and try to find a new match.

Elimination of Associative Lists In Chapters 12, 13 and 14 we will show that TRSs with associative lists can be transformed to equivalent TRSs without associative lists.

This is done by representing associative lists by cons/nil lists. To cope with list patterns like $\{X*, I, Y*, J, Z*\}$, rules are generated that explicitly try all possible assignments to the sublist variables.

This transformation tells us what we already suspected:

- Associative lists do not add computational power to term rewriting.
- A formalism with associative lists is more expressive than a formalism without associative lists; some problems can be expressed with less rules by using associative lists than by using cons/nil lists.

Before specifying the transformations, we will first define the syntax and semantics of RN2. The following section gives an introduction to this definition. Then, in Section 8.3 we will give a precise definition of the notion of transformation.

8.2 RN2

RN2 (Rule Notation 2) is a formalism for the definition of untyped conditional term rewrite systems with associative lists. RN2 has the same expressive power as ASF+SDF with respect to term rewriting. RN2 was designed by Walters and Kamperman [WK93] as intermediate language for the ASF+SDF to C compiler.

In this section we shortly discuss syntax, static and dynamic semantics of RN2. Algebraic definitions will be given in, respectively, the Chapters 9, 10, and 11.

8.2.1 Syntax

We have defined RN2 as a hierarchy of languages. The stages in the hierarchy are discussed shortly. Section 9.1 contains the syntax definitions of these languages.

BRN2 Basic RN2 is a language with terms built from variables and function application. It has unconditional rewrite rules over terms.

CRN2 Conditional RN2 adds conditional rewrite rules to BRN2. Three kinds of conditions are defined. The normal positive conditions ($\stackrel{?}{=}$) require two terms to reduce to the same normal form. Newvar or assignment conditions (:=) assign terms to the variables in the left-hand side if the lhs matches with the normal form of the instantiated rhs. Negative conditions ($\neq$) require the lhs and rhs to reduce to different normal forms.

DRN2 Default RN2 adds *default rules* to BRN2. A default rule can only be applied to a term if no other rule for the function is applicable.

LRN2 List RN2 adds associative lists to BRN2. In LRN2 associative lists are written as a sequence of terms in square brackets [and].

RN2 is the combination of CRN2, DRN2 and LRN2.

RN2 versus RN1 The ASF+SDF compiler ASFtoC uses RN1 as intermediate language. RN1 has the property that RN1 specifications can be read as RN1 terms. This makes it a straightforward matter to define specifications that manipulate specifications.

RN1 can be seen as the abstract syntax of RN2. The only difference between the two formalisms is some syntactic sugar in RN2. For instance, in RN2 a specification is written as

$$\mathit{signature} \ldots \mathit{equations} \ldots$$

while in RN1 this is written as

$$\mathit{spec}([\ldots],[\ldots])$$

The syntactic sugar makes RN2 more readable for human beings.

8.2.2 Semantics

Static Semantics In Chapter 10 we define the functions

$$tc[\![\text{TRS}]\!] \to \text{Boolean}$$

and

$$tc[\![\text{TRS}, \text{TERM}]\!] \to \text{Boolean}$$

The first function checks the static rules for RN2 TRSs. The second rule checks if an RN2 TERM is correct as a ground term over the signature of a TRS.

Dynamic Semantics In Chapter 11 we define the function

$$\text{TRS}[\![\text{TERM}]\!] \to \text{TERM}$$

that reduces a TERM t to its normal form in the context of a TRS $\mathcal{T}$. This definition includes the definition of list matching and conditional rewriting with default rules. We will assume that $\mathcal{T}$ is a confluent TRS and that the reduction of t terminates.

8.3 Transformations on RN2

In this section we will look closer at the notion of transformations on RN2.

A transformation changes a TRS to an equivalent TRS. With equivalent we mean that the transformed specification has the same semantics as the original one, for terms over the original signature. A transformation is usually a simplification in terms of the language features used.

To have a better understanding of what a transformation should do, we give the following definition in terms of the static and dynamic semantics of RN2.

Definition 8.3.1 (Transformation) A tuple of functions

$$\begin{array}{lll} tr = \langle & tr & [\![\text{TRS}]\!] \to \text{TRS}, \\ & tr & [\![\text{TERM}]\!] \to \text{TERM}, \\ & tr^{-1} & [\![\text{TERM}]\!] \to \text{TERM}, \\ & tr_{\text{pre}} & [\![\text{TRS}]\!] \to \text{Boolean}, \\ & tr_{\text{post}} & [\![\text{TRS}]\!] \to \text{Boolean} \quad \rangle \end{array}$$

is a transformation on RN2, if for any TRS $\mathcal{T}$ and for any TERM t

$$\begin{array}{l} (tc[\![\mathcal{T}]\!] \wedge tr_{\text{pre}}[\![\mathcal{T}]\!]) \to (tr_{\text{post}}[\![tr[\![\mathcal{T}]\!]]\!] \\ \qquad \wedge\, (tc[\![\mathcal{T}, t]\!] \to tr^{-1}[\![\ tr[\![\mathcal{T}]\!]\ [\![\ tr[\![t]\!]\ ]\!]\ ]\!] \equiv \mathcal{T}[\![t]\!])) \end{array}$$

This definition says that a transformation consists of five functions, the first of which is the actual TRS transforming function. The function tr_{pre} is a precondition for the transformation. It expresses to which static correct TRSs the transformation is applicable. The function tr_{post} is the postcondition of the transformation. It expresses what changes are made by the transformation. The function tr on TERMs preprocesses a TERM before it is reduced by the transformed TRS. The function tr^{-1} processes the normal form that is produced by reducing a term with the transformed TRS.

The formula in the definition says that, if a TRS $\mathcal{T}$ is statically correct and satisfies the precondition, then the transformed TRS $tr[\![\mathcal{T}]\!]$ should satisfy the postcondition and, if the term t is a closed term over the signature of $\mathcal{T}$, then the normal form of t in $\mathcal{T}$ ($\mathcal{T}[\![t]\!]$) should be equal to

$$tr^{-1}[\![\ tr[\![\mathcal{T}]\!]\ [\![\ tr[\![t]\!]\ ]\!]\ ]\!]$$

that is, the result of preprocessing t with tr, reducing with the transformed TRS and postprocessing with tr^{-1}. The pre- and postprocessing functions tr and tr^{-1} are needed because some transformations change the signature of TRSs.

The following diagram illustrates the equality condition.

$$\begin{tikzcd} t \arrow[r, "tr"] \arrow[d, "\mathcal{T}"'] & t' \arrow[d, "tr[\![\mathcal{T}]\!]"] \\ s & s' \arrow[l, "tr^{-1}"] \end{tikzcd}$$

The arrow from t to s is the reduction with the original TRS. The arrows from t via t' and s' to s show that the transformed TRS has the same effect on t as the original TRS.

In Section 15.1.1 we will use the condition in definition 8.3.1 to define a correctness test for the transformations developed in this thesis.

Since the new compiler will consist of a series of transformations that are applied successively to a TRS, we need the notion of composition of transformations.

Definition 8.3.2 (Composition of Transformations) If tr_1 and tr_2 are transformations, such that for all TRSs $\mathcal{T}$ $tr_{\mathrm{post}_1}[\![\mathcal{T}]\!] \rightarrow tr_{\mathrm{pre}_2}[\![\mathcal{T}]\!]$, then $tr_2 \circ tr_1$, the composition of tr_1 and tr_2, is the transformation

$$\begin{array}{llllll} \langle & tr_2 & [\![\mathrm{TRS}]\!] \rightarrow \mathrm{TRS} & \circ & tr_1 & [\![\mathrm{TRS}]\!] \rightarrow \mathrm{TRS}, \\ & tr_2 & [\![\mathrm{TERM}]\!] \rightarrow \mathrm{TERM} & \circ & tr_1 & [\![\mathrm{TERM}]\!] \rightarrow \mathrm{TERM}, \\ & tr_1^{-1} & [\![\mathrm{TERM}]\!] \rightarrow \mathrm{TERM} & \circ & tr_2^{-1} & [\![\mathrm{TERM}]\!] \rightarrow \mathrm{TERM}, \\ & tr_{\mathrm{pre}_1} & [\![\mathrm{TRS}]\!] \rightarrow \mathrm{Boolean} & & & \\ & tr_{\mathrm{post}_2} & [\![\mathrm{TRS}]\!] \rightarrow \mathrm{Boolean} & & & \rangle \end{array}$$

8.3.1 Requirements

To rule out unrealistic solutions for transformations we pose the following requirements on them. When we say 'transformed TRS' we mean 'the executable resulting from compiling the transformed TRS'. Similarly we abbreviate 'the executable resulting from compiling the original TRS' by 'the original TRS'.

Requirement 1 (Size of the Specification) The size of the transformed TRS should be smaller than n times the size of the original TRS, where n is a small constant.

Requirement 2 (Reduction Time) The time needed by the transformed TRS to reduce a term should be less or equal to n times the reduction time of the original TRS, where n is a small constant.

Requirement 3 (Memory Usage) The memory usage of the transformed TRS should be the same as or less than the memory usage of the original TRS.

Requirement 4 (Complexity of the Transformation) The specification of the transformation should be executable, and use a reasonable amount of time and space to execute.

8.4 Overview

Chapter 9 defines the syntax of RN2, incorporates the RN2 constructs in the language of combinatory terms, and then introduces the terminology used to manipulate RN2 objects. Chapter 10 defines the static constraints on RN2 specifications.

Chapter 11 defines the interpretation of TRSs as term reducing functions. This includes the definition of assignments (of variables to terms), (list) matching, and the (leftmost-innermost) reduction algorithm.

Chapters 12, 13 and 14 then define the transformations *lm*, *fl* and *ll*. Transformation *ll* makes specifications left-linear. Transformation *fl* flattens associative list expressions. Transformation *lm* eliminates associative lists from flat specifications. The composition of these transformations, $lm \circ fl \circ ll$, transforms a TRS with associative lists to a TRS with lists constructed by *cons* and *nil*.

In Chapter 15 a correctness test for the transformations is defined, the integration of the transformations in the compiler and the validation of the requirements on transformations are discussed. Finally, in Chapter 16 some concluding remarks are made.

Chapter 9

Syntax and Terminology

In this chapter we define the syntax of the term-rewrite formalism *Rule Notation 2* (RN2). Sections 9.1.1 through 9.1.5 define the kernel formalism Basic-RN2 and several extensions to it. In Section 9.2 we give several examples of RN2 specifications. In Section 9.3 we incorporate RN2 into our combinatory setting. In Sections 9.4.2 trough 9.4.7 we introduce terminology and define basic combinators for manipulation of RN2 specifications. Finally in Section 9.5 we define the *environments* datastructure used in transformations.

The import graphs for this chapter are shown in Figures 9.1, 9.2 and 9.3.

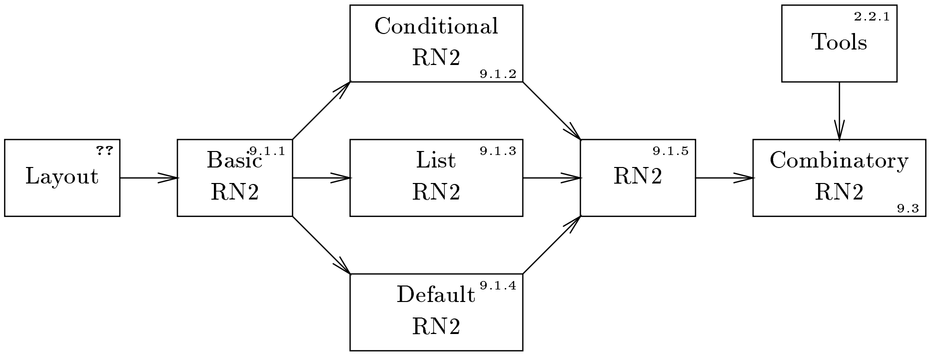


Figure 9.1: Import graph for Chapter 9; Syntax

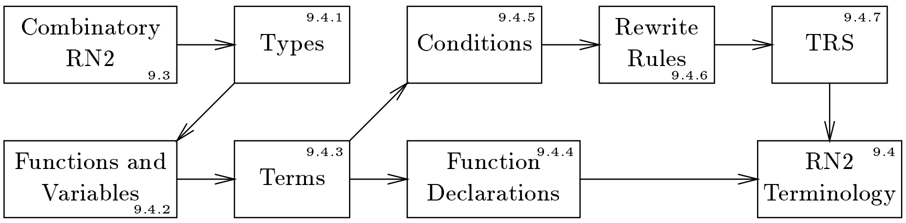


Figure 9.2: Import graph for Chapter 9; Terminology

9.1 Syntax

RN2 is the combination of three extensions of Basic RN2: CRN2 adds *conditional* rules, DRN2 adds *default* rules and LRN2 adds *associative lists*.

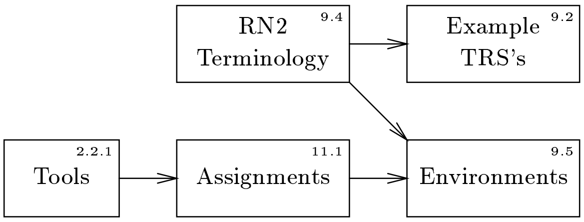


Figure 9.3: Import graph for Chapter 9; Examples and Environments

9.1.1 Basic-RN2

Imports Layout(??)

Basic-RN2 is a simple specification formalism for the definition of untyped, unconditional TRSs. A specification consists of a signature and a list of rewrite rules (or equations). The signature declares functions and their arity. The type *appTp* serves in the declaration of the arity of functions. Terms are built from variables and applications of functions. A function without arguments is called a constant.

The lexical syntax of function symbols and variables allow functions starting with a *$* sign and variables containing one or more *$* signs. Specifications that are to be compiled should not contain such functions and variables. This simplifies the generation of names of functions and variables.

Exports

Sorts FUNSYM TYPE FUNDECL APVAR VAR TERM RULE TRS

Lexical Syntax

[a-z$] [a-zA-Z0-9]∗ → FUNSYM
[A-Z] [a-zA-Z0-9$]∗ → APVAR

Context-free Functions

appTp	→ TYPE
FUNSYM "(" {TYPE "," }∗ ")"	→ FUNDECL
APVAR	→ VAR
VAR	→ TERM
FUNSYM "(" {TERM "," }∗ ")"	→ TERM
"[" FUNSYM "]" TERM "=" TERM	→ RULE
"*signature*" FUNDECL∗ "*equations*" RULE∗	→ TRS

Variables

"$\mathcal{T}$" *[0-9′]∗*	→ TRS
"*fd*" *[0-9′]∗*	→ FUNDECL
"$\vec{fd}$" *[0-9′]∗*	→ FUNDECL∗
"τ" *[0-9′]∗*	→ TYPE
"$\vec{\tau}$" *[0-9′]∗*	→ {TYPE "," }∗
"φ" *[0-9′]∗*	→ RULE
"$\vec{\varphi}$" *[0-9′]∗*	→ RULE∗
"*a*" *[0-9′]∗*	→ TERM
"$\vec{a}$" *[0-9′]∗*	→ {TERM "," }∗
"*b*" *[0-9′]∗*	→ TERM
"$\vec{b}$" *[0-9′]∗*	→ {TERM "," }∗
"*c*" *[0-9′]∗*	→ TERM
"$\vec{c}$" *[0-9′]∗*	→ {TERM "," }∗
"*d*" *[0-9′]∗*	→ TERM

“$\vec{d}$” $[0\text{-}9']*$ → {TERM “,” }*
“e” $[0\text{-}9']*$ → TERM
“$\vec{e}$” $[0\text{-}9']*$ → {TERM “,” }*
“s” $[0\text{-}9']*$ → TERM
“$\vec{s}$” $[0\text{-}9']*$ → {TERM “,” }*
“t” $[0\text{-}9']*$ → TERM
“$\vec{t}$” $[0\text{-}9']*$ → {TERM “,” }*

“f” $[0\text{-}9']*$ → FUNSYM
“g” $[0\text{-}9']*$ → FUNSYM

“x” $[0\text{-}9']*$ → APVAR
“y” $[0\text{-}9']*$ → APVAR
“$x^{(*)}$” $[0\text{-}9']*$ → VAR
“$y^{(*)}$” $[0\text{-}9']*$ → VAR

9.1.2 Conditional-RN2

Imports BRN2$^{(9.1.1)}$

We add conditions and conditional rewrite rules to BRN2. Conditions come in three flavours. A positive condition ($\stackrel{?}{=}$) requires its left-hand side and right-hand side to be equal. A negative condition ($\neq$) requires unequality. An *assignment* or *newvars* condition (:=) assigns it rhs to its lhs, instantiating the variables of the lhs. A conditional rule is a rule with one or more conditions.

Exports
Sorts BAR CONDITION
Lexical Syntax
“===” [=]* [>]* → BAR
Context-free Functions
TERM “$\stackrel{?}{=}$” TERM → CONDITION
TERM “$\neq$” TERM → CONDITION
TERM “:=” TERM → CONDITION

“[” FUNSYM “]”
{CONDITION “,” }+ BAR
TERM “=” TERM → RULE
Variables
“$\Rightarrow$” $[0\text{-}9']*$ → BAR
“C” $[0\text{-}9']*$ → CONDITION
“$\vec{C}$” $[0\text{-}9']*$ → {CONDITION “,” }*
“C^{+}” $[0\text{-}9']*$ → {CONDITION “,” }+

9.1.3 List-RN2

Imports BRN2$^{(9.1.1)}$

LRN2 extends BRN2 with associative lists. An associative list is a special function with a variable number of arguments. A special kind of variable, called sublist variable, is introduced that can appear only as argument of lists, and that denotes an arbitrary sublist of the list. The new type *lstTp* is used in function declarations to indicate that a function may have a list as argument.

Exports
Sorts SLVAR
Lexical Syntax
[A-Z] [a-zA-Z0-9$]** [*]+ → SLVAR

Context-free Functions

$lstTp$ → TYPE
SLVAR → VAR
"[" {TERM "," }∗ "]" → TERM

Variables

"x*" [*0-9′*]∗ → SLVAR
"y*" [*0-9′*]∗ → SLVAR

9.1.4 Default-RN2

Imports $\mathrm{BRN2}^{(9.1.1)}$

In this module we add *default* rules to BRN2. A default rule can only be used for rewriting after all normal rules in the TRS have been tried. That is, default rules have a lower priority than normal rules.

Exports

Context-free Functions

otherwise ":" RULE → RULE

Equations

We only consider one level of default-ness. A default default rule is just a default rule.

$$otherwise : otherwise : \varphi = otherwise : \varphi$$

9.1.5 RN2

Imports $\mathrm{CRN2}^{(9.1.2)}$ $\mathrm{DRN2}^{(9.1.4)}$ $\mathrm{LRN2}^{(9.1.3)}$

The language RN2 is the extension of Basic-RN2 with the three extensions CRN2, DRN2 and LRN2. Henceforth we will refer to other combinations of these extensions by prefixing their first letter in front of 'RN2'; the language CD-RN2 is BRN2 with conditional and default rules.

9.2 Example TRSs

Imports $\text{RN2-Terminology}^{(9.4)}$

In this module we give some examples of RN2 TRSs. The function *example* maps (some) naturals to TRSs. Example TRSs may be combined by applying *example* to a list of naturals. In following chapters we will refer to these examples as $example[\![n]\!]$. In module 15.1.1 we will define functions that test the correctness of the transformations, using these examples.

Exports

Context-free Functions

example "⟦" {NAT "," }∗ "⟧" → TRS

Equations

Example specifications are combined by the operator +, which will be defined in module 9.4 for the addition of TRSs.

$$example[\![n, n^+]\!] = \mathcal{T} \Leftarrow \mathcal{T} = example[\![n]\!] + example[\![n^+]\!]$$

Constants Examples 0 defines some constants for use in testing.

$$example[\![0]\!] = signature\ a()\ b()\ c()\ d()\ e()\ f()\ g()\ h()\ i()\ j()\ k()\ equations$$

Example 1 defines the Boolean constants *yes* and *no*.

$$example[\![1]\!] = signature\ yes()\ no()\ equations$$

BRN2 Example 2 shows the specification of successor naturals with *plus*.

$$\begin{aligned} example[\![2]\!] = {} & signature\ zero()\ succ(appTp)\ plus(appTp,\ appTp) \\ & equations\ [plus0]\ plus(zero(),\ X) = X \\ & \qquad\qquad [plusn]\ plus(succ(X),\ Y) = succ(plus(X,\ Y)) \end{aligned}$$

CRN2 Example 3 defines the equality predicate *eq* using a positive and negative condition.

$$\begin{aligned} example[\![3]\!] = {} & example[\![1]\!] \\ & + signature\ eq(appTp,\ appTp) \\ & \quad equations\ [eq]\ A \stackrel{?}{=} B ===> eq(A,\ B) = yes() \\ & \qquad\qquad\quad [eq]\ A \neq B ===> eq(A,\ B) = no() \end{aligned}$$

DRN2 The equality predicate from $example[\![3]\!]$ can also be specified with default rules as follows.

$$\begin{aligned} example[\![4]\!] = {} & example[\![1]\!] \\ & + signature\ eq(appTp,\ appTp) \\ & \quad equations\ [eqy]\ eq(A,\ A) = yes() \\ & \qquad\qquad\quad otherwise : [eqn]\ eq(A,\ B) = no() \end{aligned}$$

The first rule states that if *eq* is applied to two equal terms they are equal. The second rule applies if the first rule is not applicable; A and B have different normal forms and thus are unequal.

LRN2 The following examples are TRSs with rules over associative lists. The examples show rules of increasing difficulty with respect to the transformation that eliminates lists.

The function *sum* in $example[\![5]\!]$ adds the successor naturals in a list. The lhs of the second rule uses the pattern $[N, N*]$ to divide the list in its head N and its tail $[N*]$.

$$\begin{aligned} example[\![5]\!] = {} & example[\![2]\!] \\ & + signature\ sum(lstTp) \\ & \quad equations\ [sum]\ sum([]) = zero() \\ & \qquad\qquad\quad [sum]\ sum([N,\ N*]) = plus(N,\ sum([N*])) \end{aligned}$$

The function *revers* reverses a list. The second rule uses the pattern $[X*, X]$ to get the last element of the list X and the initial segment $[X*]$.

$$\begin{aligned} example[\![6]\!] = {} & signature\ revers(lstTp) \\ & equations\ [rev0]\ revers([]) = [] \\ & \qquad\qquad [revn]\ [Y*] := revers([X*]) \\ & \qquad\qquad\qquad ===> revers([X*,\ X]) = [X,\ Y*] \end{aligned}$$

The function *palindrome* checks wether the list it is applied to is a palindrome. That is, a list that is the concatenation of a list and its reverse, possibly seperated by an element. The third rule states that the first and last elements of the list should be equal. If they are not equal the result is $no()$.

$$
\begin{aligned}
example[\![7]\!] \;=\; & example[\![1]\!] \\
& + \textit{signature } palindrome(lstTp) \\
& \quad \textit{equations } [pal0]\ palindrome([]) = yes() \\
& \qquad\qquad\quad\ [pal1]\ palindrome([A]) = yes() \\
& \qquad\qquad\quad\ [paln]\ palindrome([A,\ A*,\ A]) = palindrome([A*]) \\
& \qquad\qquad\quad\ [paln]\ A \neq B \\
& \qquad\qquad\qquad\quad ====> palindrome([A,\ A*,\ B]) = no()
\end{aligned}
$$

The function *set* removes duplicates from a list. The pattern $[X*, I, Y*, I, Z*]$ matches any list with two equal elements. This TRS will be the subject of section 12.2 to illustrate the transformation to TRSs without associative lists.

$$
\begin{aligned}
example[\![8]\!] \;=\; & \textit{signature } set(lstTp) \\
& \textit{equations } [set]\ set([X*,\ I,\ Y*,\ I,\ Z*]) = set([X*,\ I,\ Y*,\ Z*])
\end{aligned}
$$

The following example shows the strength of associative lists combined with default rules. The positive rule for the *member* function is done with a pattern that (implicitly) scans a list searching for X. If that fails the default rule applies and the answer is $no()$.

$$
\begin{aligned}
example[\![9]\!] \;=\; & example[\![1]\!] \\
& + \textit{signature } member(appTp,\ lstTp) \\
& \quad \textit{equations } [memb1]\ member(X,\ [A*,\ X,\ B*]) = yes() \\
& \qquad\qquad\quad\ \textit{otherwise} : [memb2]\ member(X,\ Y) = no()
\end{aligned}
$$

In Example 10 a graph is represented by a list of nodes $p(X, [X*])$ that declare the elements of $[X*]$ as neighbours of X. The function *trans* makes the transitive closure of a graph. If Y is a neighbour of X and Z is a neighbour of Y, then Z is a neighbour of X. The second condition prevents the function from looping.

$$
\begin{aligned}
example[\![10]\!] \;=\; & example[\![8,\ 9]\!] \\
& + \textit{signature } p(appTp,\ lstTp)\ trans(lstTp) \\
& \quad \textit{equations } [trans] \\
& \qquad\qquad [P3*,\ p(Y2,\ [B1*,\ Z,\ B2*]),\ P4*] := [P1*,\ P2*], \\
& \qquad\qquad Y1 \stackrel{?}{=} Y2, \\
& \qquad\qquad member(Z,\ [A1*,\ Y1,\ A2*]) \stackrel{?}{=} no(), \\
& \qquad\qquad set([X*]) := set([A1*,\ Y1,\ A2*,\ B1*,\ Z,\ B2*]) \\
& \qquad\qquad ===> \\
& \qquad\qquad trans([P1*,\ p(X,\ [A1*,\ Y1,\ A2*]),\ P2*]) = trans([P1*,\ p(X,\ [X*]),\ P2*])
\end{aligned}
$$

In the last example the function *join* is defined. It unites the sets in a list of sets, if they share a common element.

$$
\begin{aligned}
example[\![11]\!] \;=\; & example[\![8]\!] \\
& + \textit{signature } join(lstTp) \\
& \quad \textit{equations } [join]\ join([A1*,\ set([X1*,\ I,\ X2*]),\ A2*,\ set([Y1*,\ I,\ Y2*]),\ A3*]) \\
& \qquad\qquad\quad = join([A1*,\ set([X1*,\ I,\ X2*,\ Y1*,\ Y2*]),\ A2*,\ A3*])
\end{aligned}
$$

9.3 Combinatory RN2

Imports $\text{RN2}^{(9.1.5)}$ $\text{Tools}^{(2.2.1)}$

We embed the sorts of the RN2 syntax in our combinatory language and define the primitive list operations for the lists sorts.

Exports

Context-free Functions

TRS	$\to$ Constant
FUNDECL	$\to$ Constant
TYPE	$\to$ Constant
RULE	$\to$ Constant
CONDITION	$\to$ Constant
TERM	$\to$ Constant
FUNSYM	$\to$ Constant

We define new combinatory list sorts for lists of function declarations, types, rules conditions and terms.

Exports

Context-free Functions

"$\llbracket$" FUNDECL$*$ "$\rrbracket_{\vec{fd}}$"	$\to$ Constant
"$\llbracket$" {TYPE "," }$*$ "$\rrbracket_{\vec{\tau}}$"	$\to$ Constant
"$\llbracket$" RULE$*$ "$\rrbracket_{\vec{\varphi}}$"	$\to$ Constant
"$\llbracket$" {CONDITION "," }$*$ "$\rrbracket_{\vec{C}}$"	$\to$ Constant
"$\llbracket$" {TERM "," }$*$ "$\rrbracket_{\vec{t}}$"	$\to$ Constant

For conversion of lists to the list sorts above we define the following combinators.

Exports

Context-free Functions

"$\downarrow_{\vec{fd}}$"	$\to$ Combinator
"$\downarrow_{\vec{\tau}}$"	$\to$ Combinator
"$\downarrow_{\vec{\varphi}}$"	$\to$ Combinator
"$\downarrow_{\vec{C}}$"	$\to$ Combinator
"$\downarrow_{\vec{t}}$"	$\to$ Combinator

Equations

Conversion Combinators

$$\downarrow_{\vec{fd}} = \downarrow \llbracket\ \rrbracket_{\vec{fd}} \qquad \downarrow_{\vec{\tau}} = \downarrow \llbracket\rrbracket_{\vec{\tau}} \qquad \downarrow_{\vec{\varphi}} = \downarrow \llbracket\rrbracket_{\vec{\varphi}} \qquad \downarrow_{\vec{C}} = \downarrow \llbracket\rrbracket_{\vec{C}} \qquad \downarrow_{\vec{t}} = \downarrow \llbracket\rrbracket_{\vec{t}}$$

List Operations The obvious definition of the primitive list operations for lists of function declarations, types, rules, conditions, and terms.

$$\begin{array}{llll}
hd\ \llbracket fd\ \vec{fd}\ \rrbracket_{\vec{fd}} = fd & & tl\ \llbracket fd\ \vec{fd}\ \rrbracket_{\vec{fd}} & = \llbracket \vec{fd}\ \rrbracket_{\vec{fd}} \\
\varepsilon?\ \llbracket\ \rrbracket_{\vec{fd}} & = \top & \varepsilon?\ \llbracket fd\ \vec{fd}\ \rrbracket_{\vec{fd}} = \bot & \\
nil\ \llbracket \vec{fd}\ \rrbracket_{\vec{fd}} & = \llbracket\ \rrbracket_{\vec{fd}} & fd : \llbracket \vec{fd}\ \rrbracket_{\vec{fd}} & = \llbracket fd\ \vec{fd}\ \rrbracket_{\vec{fd}}
\end{array}$$

$$\begin{array}{llll}
hd\ \llbracket \tau, \vec{\tau} \rrbracket_{\vec{\tau}} = \tau & & tl\ \llbracket \tau, \vec{\tau} \rrbracket_{\vec{\tau}} & = \llbracket \vec{\tau} \rrbracket_{\vec{\tau}} \\
\varepsilon?\ \llbracket\rrbracket_{\vec{\tau}} & = \top & \varepsilon?\ \llbracket \tau, \vec{\tau} \rrbracket_{\vec{\tau}} = \bot & \\
nil\ \llbracket \vec{\tau} \rrbracket_{\vec{\tau}} & = \llbracket\rrbracket_{\vec{\tau}} & \tau : \llbracket \vec{\tau} \rrbracket_{\vec{\tau}} & = \llbracket \tau, \vec{\tau} \rrbracket_{\vec{\tau}}
\end{array}$$

$$\begin{array}{llll}
hd\ \llbracket \varphi\ \vec{\varphi} \rrbracket_{\vec{\varphi}} = \varphi & & tl\ \llbracket \varphi\ \vec{\varphi} \rrbracket_{\vec{\varphi}} & = \llbracket \vec{\varphi} \rrbracket_{\vec{\varphi}} \\
\varepsilon?\ \llbracket\rrbracket_{\vec{\varphi}} & = \top & \varepsilon?\ \llbracket \varphi\ \vec{\varphi} \rrbracket_{\vec{\varphi}} = \bot & \\
nil\ \llbracket \vec{\varphi} \rrbracket_{\vec{\varphi}} & = \llbracket\rrbracket_{\vec{\varphi}} & \varphi : \llbracket \vec{\varphi} \rrbracket_{\vec{\varphi}} & = \llbracket \varphi\ \vec{\varphi} \rrbracket_{\vec{\varphi}}
\end{array}$$

$$
\begin{array}{llll}
hd\,[\![C,\vec{C}]\!]_{\vec{C}} = C & tl\,[\![C,\vec{C}]\!]_{\vec{C}} = [\![\vec{C}]\!]_{\vec{C}} \\
\varepsilon?\,[\![]\!]_{\vec{C}} = \top & \varepsilon?\,[\![C,\vec{C}]\!]_{\vec{C}} = \bot \\
nil\,[\![\vec{C}]\!]_{\vec{C}} = [\![]\!]_{\vec{C}} & C : [\![\vec{C}]\!]_{\vec{C}} = [\![C,\vec{C}]\!]_{\vec{C}}
\end{array}
$$

$$
\begin{array}{ll}
hd\,[\![t,\vec{t}]\!]_{\vec{t}} = t & tl\,[\![t,\vec{t}]\!]_{\vec{t}} = [\![\vec{t}]\!]_{\vec{t}} \\
\varepsilon?\,[\![]\!]_{\vec{t}} = \top & \varepsilon?\,[\![t,\vec{t}]\!]_{\vec{t}} = \bot \\
nil\,[\![\vec{t}]\!]_{\vec{t}} = [\![]\!]_{\vec{t}} & t : [\![\vec{t}]\!]_{\vec{t}} = [\![t,\vec{t}]\!]_{\vec{t}}
\end{array}
$$

9.4 RN2 Terminology

Imports Functions-and-Variables$^{(9.4.2)}$ Types$^{(9.4.1)}$ Terms$^{(9.4.3)}$ Function-Declarations$^{(9.4.4)}$ Conditions$^{(9.4.5)}$ Rewrite-Rules$^{(9.4.6)}$ TRS$^{(9.4.7)}$

This module combines modules that define combinators for the manipulation of RN2 constructs. The modules also introduce some common terminology with respect to TRSs.[1]

9.4.1 Types

Imports Combinatory-RN2$^{(9.3)}$

Although RN2 is an untyped language, a limited notion of types is used in specifications. The first use of types is the distinction between functions that operate on applications only and functions that also may have lists as arguments. For this purpose the types *appTp* and *lstTp* are used in signatures. The second use is in checking the static rules on RN2 specifications. The third use is distinguishing the four kinds of TERMs. For the latter purposes the types *varTp* and *slTp* are introduced in this module. The combinator *type* gives the type of a term.

We impose a linear order on types. Terms with types lower in the order may appear in places where terms with types higher in the order may appear. For instance, normal variables may appear as argument of any function, while sublist variables may appear only as arguments of lists.

Exports

Context-free Functions

slTp $\to$ TYPE
varTp $\to$ TYPE
type $\to$ Combinator {**type:** TERM $\to$ TYPE}
"$\preceq$" $\to$ Infix-Operator {**type:** TYPE $\times$ TYPE $\to \mathbb{B}$}
"$\succeq$" $\to$ Infix-Operator {**type:** TYPE $\times$ TYPE $\to \mathbb{B}$}

Hiddens

Context-free Functions

ord $\to$ Combinator {**type:** TYPE $\to n$}

Equations

Types of Terms

$$
\begin{array}{ll}
type\ x = varTp & type\ x^* = slTp \\
type\ f(\vec{t}) = appTp & type\ [\vec{t}] = lstTp
\end{array}
$$

[1] The fact that the following modules are presented as subsections of this section does not mean they are locally defined; in ASF+SDF there is no notion of local or nested modules.

Subtype and Supertype The rules for the subtype relation are: Any type is a subtype of itself, the variable type is a subtype of the list type and of the application type, application is a subtype of list, and any type is a subtype of sublist type. In short $\preceq$ is the following order on types

$$varTp \ \preceq \ appTp \ \preceq \ lstTp \ \preceq \ slTp$$

Clearly this is a linear order so we can implement the $\preceq$ relation by converting types (taking their *ord*) to naturals and comparing them with the $\leq$ relation on naturals,

$$\tau_1 \preceq \tau_2 = ord\, \tau_1 \leq ord\, \tau_2$$

where *ord* is defined as

$$ord\ varTp = 0 \qquad ord\ appTp = 1 \qquad ord\ lstTp = 2 \qquad ord\ slTp = 3$$

The generalisation of the subtype relation for lists of types is then specified using *compare*[2] as

$$[\![\vec{\tau}_1]\!]_{\vec{\tau}} \preceq [\![\vec{\tau}_2]\!]_{\vec{\tau}} = compare\ (\preceq)\ [\![\vec{\tau}_1]\!]_{\vec{\tau}}\ [\![\vec{\tau}_2]\!]_{\vec{\tau}}$$

The supertype relation ($\succeq$) is the reflexive closure of the complement of $\preceq$.

$$\tau_1 \succeq \tau_2 = \tau_2 \preceq \tau_1 \qquad [\![\vec{\tau}_1]\!]_{\vec{\tau}} \succeq [\![\vec{\tau}_2]\!]_{\vec{\tau}} = [\![\vec{\tau}_2]\!]_{\vec{\tau}} \preceq [\![\vec{\tau}_1]\!]_{\vec{\tau}}$$

9.4.2 Functions and Variables

Imports Types$^{(9.4.1)}$

We define normal and sublist variables as lists of characters. Furthermore combinators for manipulation of function symbols and variables are defined. *Var* constructs a variable from a string. The postfix-operators $*$ and $\backslash*$ convert normal variables to sublist variables and sublist variables to normal variables, respectively. The operator _ is an indexing function for functions and variables.

Exports

Context-free Functions

"*Var*"	$\to$ Combinator	{**type:** Character* $\to$ APVAR}
"$*$"	$\to$ Postfix-Operator	{**type:** APVAR $\to$ SLVAR}
"$\backslash*$"	$\to$ Postfix-Operator	{**type:** SLVAR $\to$ APVAR}
"_"	$\to$ Infix-Operator	{**type:** (VAR $\times\ \mathbb{N} \to$ VAR) $\cup$ (FUNSYM $\times\ \mathbb{N} \to$ FUNSYM)}

Hiddens

Variables

[*X*]	$\to$ Term
[*c*] [*0-9′*]$*$	$\to$ CHAR
[*c*] [$*$] [*0-9′*]$*$	$\to$ CHAR$*$
[*c*] [+] [*0-9′*]$*$	$\to$ CHAR+

Equations

Variables as Lists of Characters

$$character(\text{"‘"}\ \ c\ \text{"'"}\ ) : nil\ x \qquad = apvar(c)$$
$$character(\text{"‘"}\ \ c\ \text{"'"}\ ) : apvar(c+) = apvar(c\ c+)$$

[2] See Section 5.6

$$\begin{array}{llll} hd\ apvar(c\ c*) & = character(\text{“‘”}\quad c\ \text{“'”}\ ) & & \\ tl\ apvar(c\ c+) & = apvar(c+) & tl\ apvar(c) & = nil\ apvar(c) \\ \varepsilon?\ x & = \bot & \varepsilon?\ (nil\ x) & = \top \end{array}$$

$$\begin{array}{ll} character(\text{“‘”}\quad c\ \text{“'”}\ ) : nil\ x^* & = slvar(c) \\ character(\text{“‘”}\quad c\ \text{“'”}\ ) : slvar(c+) & = slvar(c\ c+) \end{array}$$

$$\begin{array}{llll} hd\ slvar(c\ c*) & = character(\text{“‘”}\quad c\ \text{“'”}\ ) & & \\ tl\ slvar(c\ c+) & = slvar(c+) & tl\ slvar(c) & = nil\ slvar(c) \\ \varepsilon?\ x^* & = \bot & \varepsilon?\ (nil\ x^*) & = \top \end{array}$$

Functions as Lists of Characters

$$\begin{array}{ll} character(\text{“‘”}\quad c\ \text{“'”}\ ) : nil\ f & = funsym(c) \\ character(\text{“‘”}\quad c\ \text{“'”}\ ) : funsym(c+) & = funsym(c\ c+) \end{array}$$

$$\begin{array}{llll} hd\ funsym(c\ c*) & = character(\text{“‘”}\quad c\ \text{“'”}\ ) & & \\ tl\ funsym(c\ c+) & = funsym(c+) & tl\ funsym(c) & = nil\ funsym(c) \\ \varepsilon?\ f & = \bot & \varepsilon?\ (nil\ f) & = \top \end{array}$$

String to Variable *Var* converts lists of characters (strings for instance) to variables.

$$Var = \downarrow (nil\ apvar(\text{“}X\text{”}\ ))$$

APVAR to SLVAR and Back The postfix-operator $*$ converts normal variables to sublist variables. The postfix-operator $\backslash *$ converts sublist variables to normal variables.

$$\begin{array}{llll} apvar(c+)\ * & = slvar(c+\ \text{“}*\text{”}\ ) & t\ * & = t \Leftarrow type\ t \neq varTp \\ slvar(c+\ \text{“}*\text{”}\ )\ \backslash * & = apvar(c+\ \text{“\$”}\ ) & t\ \backslash * & = t \Leftarrow type\ t \neq slTp \end{array}$$

Indexing Function Symbols and Variables The infix-operator (_) constructs an indexed variable or function from a variable or function and a natural.

$$\begin{array}{ll} apvar(c+)\ \_\ nat\_con(c+') & = apvar(c+\ \text{“\$”}\quad c+') \\ slvar(c+)\ \_\ nat\_con(c+') & = slvar(c+\ \text{“\$”}\quad c+'\ \text{“}*\text{”}\ ) \\ x^{(*)}\ \_\ [\vec{n}] & = \downarrow_{\vec{t}} (map\ (x^{(*)}\ \_)\ [\vec{n}]) \end{array}$$

$$\begin{array}{ll} funsym(c+)\ \_\ nat\_con(c+') & = funsym(c+\ c+') \\ f\ \_\ [\vec{n}] & = map\ (f\ \_)\ [\vec{n}] \end{array}$$

Applied to lists of variables (_) constructs the list of corresponding indexed variables.

$$\begin{array}{ll} [\![\vec{t}]\!]_{\vec{t}}\ \_\ n & = map\ (\_\ n)\ [\![\vec{t}]\!]_{\vec{t}} \\ [\![\vec{t}]\!]_{\vec{t}}\ \_\ [\vec{n}] & = \downarrow_{\vec{t}} (zip\text{-}with\ (\_)\ [\![\vec{t}]\!]_{\vec{t}}\ [\vec{n}]) \end{array}$$

9.4.3 Terms

Imports Functions-and-Variables$^{(9.4.2)}$

The fact that terms can be represented as trees is expressed by defining the primitive tree operations for terms. We define synonyms for the tree construction and decomposition functions and predicates for analysis of terms.

Exports

Context-free Functions

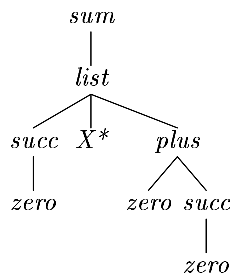
app	$\rightarrow$ Combinator {**type:** FUNSYM $\times$ TERM* $\rightarrow$ TERM}
args	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM*}
fun	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ FUNSYM}
list	$\rightarrow$ Combinator {**type:** TERM* $\rightarrow$ TERM}
items	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM*}
"*app?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*var?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*list?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*slVar?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*appVar?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*slTp?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
"*appTp?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}
subterms	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM*}
vars	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ VAR*}
funs	$\rightarrow$ Combinator {**type:** TERM $\rightarrow$ FUNSYM*}
arity	$\rightarrow$ Combinator {**type:** TERM $\rightarrow n$}
"*ground?*"	$\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{B}$}

Equations

Terms are Trees The primitive tree operations for terms:

$$
\begin{array}{lll}
\mathit{label}\ x^{(*)} = x^{(*)} & \mathit{sons}\ x^{(*)} = [\![]\!]_{\vec{t}} & \mathit{tree}\ x^{(*)}\ [\![]\!]_{\vec{t}} = x^{(*)} \\
\mathit{label}\ f(\vec{t}) = f & \mathit{sons}\ f(\vec{t}) = [\![\vec{t}]\!]_{\vec{t}} & \mathit{tree}\ f\ [\![\vec{t}]\!]_{\vec{t}} = f(\vec{t}) \\
\mathit{label}\ [\vec{t}] = \mathit{list} & \mathit{sons}\ [\vec{t}] = [\![\vec{t}]\!]_{\vec{t}} & \mathit{tree}\ \mathit{list}\ [\![\vec{t}]\!]_{\vec{t}} = [\vec{t}]
\end{array}
$$

Example The term $\mathit{sum}([\mathit{succ}(\mathit{zero}()), X*, \mathit{plus}(\mathit{zero}(), \mathit{succ}(X))])$ can be represented by the tree



Synonyms for *tree*, *label* and *sons*. $f(\vec{t})$ is called an *app*lication of the *fun*ction f to the *arg*ument*s* $\vec{t}$. $[\vec{t}]$ is a *list* with *items* $\vec{t}$.

$$
\begin{array}{ll}
\mathit{app}\ f\ [\![\vec{t}]\!]_{\vec{t}} = f(\vec{t}) & \mathit{list}\ [\![\vec{t}]\!]_{\vec{t}} = [\vec{t}] \\
\mathit{args}\ f(\vec{t}) = [\![\vec{t}]\!]_{\vec{t}} & \mathit{items}\ [\vec{t}] = [\![\vec{t}]\!]_{\vec{t}} \\
\mathit{fun}\ f(\vec{t}) = f &
\end{array}
$$

Analyzing Terms

$$\begin{array}{llll} \mathit{app?}\ t & = \mathit{type}\ t \equiv \mathit{appTp} & \mathit{list?}\ t = \mathit{type}\ t \equiv \mathit{lstTp} \\ \mathit{slVar?}\ t & = \mathit{type}\ t \equiv \mathit{slTp} & \mathit{var?}\ t = \mathit{type}\ t \equiv \mathit{varTp} \vee (\mathit{type}\ t \equiv \mathit{slTp}) \end{array}$$

Subterms is a synonym for *subtrees*.

$$\mathit{subterms}\ t = \mathit{subtrees}\ t$$

The variables of a term are all subterms that are variables.

$$\mathit{vars}\ t = \mathit{filter}\ \mathit{var?}\ (\mathit{subterms}\ t)$$

The function symbols of a term are the function symbols of all applications in the term.

$$\mathit{funs}\ t = \mathit{map}\ \mathit{fun}\ (\mathit{filter}\ \mathit{app?}\ (\mathit{subterms}\ t))$$

The *arity* of an application is the number of arguments it is applied to.

$$\mathit{arity}\ f(\vec{t}) = \#\ [\![\vec{t}]\!]_{\vec{t}}$$

A term is *ground* if it contains no variables.

$$\mathit{ground?}\ t = \varepsilon?\ (\mathit{vars}\ t)$$

A term containing variables is *open*.

9.4.4 Function Declarations

Imports Terms$^{(9.4.3)}$

A function declaration serves to define a function symbol, its arity and (in LRN2) which arguments may be lists. The combinator *fun-decl* constructs function declarations.

Exports
 Context-free Functions
 fun-decl $\rightarrow$ Combinator {**type:** FUNSYM $\rightarrow$ TYPE* $\rightarrow$ FUNDECL}

Equations

Fun-decl constructs a function declaration from a function symbol and a list of types. The combinators *arity*, *fun* and *args* as on applications.

$$\begin{array}{llll} \mathit{fun\text{-}decl}\ f\ [\![\vec{\tau}]\!]_{\vec{\tau}} & = f(\vec{\tau}) & \mathit{fun}\ f(\vec{\tau}) & = f \\ \mathit{arity}\ f(\vec{\tau}) & = \#\ [\![\vec{\tau}]\!]_{\vec{\tau}} & \mathit{args}\ f(\vec{\tau}) & = [\![\vec{\tau}]\!]_{\vec{\tau}} \end{array}$$

A function declaration is a table entry.

$$\mathit{key}\ f(\vec{\tau}) = f \qquad \mathit{value}\ f(\vec{\tau}) = [\![\vec{\tau}]\!]_{\vec{\tau}}$$

This means that lists of function declarations (signatures) are tables and can be queried with the table operations from module 5.10.

9.4.5 Conditions

Imports Terms$^{(9.4.3)}$

We define combinators and operators to construct, decompose and analyze conditions.

Exports

Context-free Functions

cond	→ Combinator	{**type:** TERM → TERM → CONDITION}
"**:=**"	→ Infix-Operator	{**type:** TERM × TERM → CONDITION}
"$\boldsymbol{\neq}$"	→ Infix-Operator	{**type:** TERM × TERM → CONDITION}
"$\boldsymbol{\stackrel{?}{=}}$"	→ Infix-Operator	{**type:** TERM × TERM → CONDITION}
lhs	→ Combinator	{**type:** CONDITION → TERM}
rhs	→ Combinator	{**type:** CONDITION → TERM}
kind	→ Combinator	{**type:** CONDITION → $\{\mathrm{new}, \mathrm{pos}, \mathrm{neg}\}$}
pos	→ Constant	
neg	→ Constant	
new	→ Constant	
"*pos?*"	→ Combinator	{**type:** CONDITION → $\mathbb{B}$}
"*neg?*"	→ Combinator	{**type:** CONDITION → $\mathbb{B}$}
"*new?*"	→ Combinator	{**type:** CONDITION → $\mathbb{B}$}

Equations

The combinator *cond* constructs a condition from a condition kind and two terms. The kinds are *pos* (positive), *neg* (negative) and *new* (new, newvars or assignment condition). The operators $\boldsymbol{\stackrel{?}{=}}$, $\boldsymbol{\neq}$ and **:=** are synonyms for, respectively, *cond pos*, *cond neg* and *cond new*.

$$\begin{array}{llll} cond\ pos\ s\ t = s \stackrel{?}{=} t & & s \boldsymbol{\stackrel{?}{=}} t = s \stackrel{?}{=} t \\ cond\ neg\ s\ t = s \neq t & & s \boldsymbol{\neq} t = s \neq t \\ cond\ new\ s\ t = s := t & & s \mathbin{\boldsymbol{:=}} t = s := t \end{array}$$

Note that in the equations on the right the symbol $\boldsymbol{\stackrel{?}{=}}$ is an infix-operator, while $\stackrel{?}{=}$ is an RN2 constructor. The symbols on the left are the bold versions of the symbols on the right.

The combinators *kind*, *lhs* and *rhs* yield the kind, left-hand side and right-hand side of a condition.

$$\begin{array}{lll} kind\ s \stackrel{?}{=} t = pos & lhs\ s \stackrel{?}{=} t = s & rhs\ s \stackrel{?}{=} t = t \\ kind\ s \neq t = neg & lhs\ s \neq t = s & rhs\ s \neq t = t \\ kind\ s := t = new & lhs\ s := t = s & rhs\ s := t = t \end{array}$$

The predicates *pos?*, *neg?* and *new?* decide whether their argument is a positive, negative or new condition.

$$pos?\ C = kind\ C \equiv pos \qquad neg?\ C = kind\ C \equiv neg \qquad new?\ C = kind\ C \equiv new$$

9.4.6 Rewrite Rules

Imports Conditions$^{(9.4.5)}$

We define combinators to construct and decompose rewrite rules. A priority relation is imposed on rules. This order determines which rule is tried first in rewriting.

Exports

Context-free Functions

rule	$\rightarrow$ Combinator	{**type:** RULE $\rightarrow$ FUNSYM $\rightarrow$ CONDITION*} $\rightarrow$ TERM $\rightarrow$ TERM $\rightarrow$ RULE
"*norm*"	$\rightarrow$ Constant	{**type:** RULE}
"*def*"	$\rightarrow$ Constant	{**type:** RULE}
tag	$\rightarrow$ Combinator	{**type:** RULE $\rightarrow$ FUNSYM}
conds	$\rightarrow$ Combinator	{**type:** RULE $\rightarrow$ CONDITION*}
terms	$\rightarrow$ Combinator	{**type:** RULE $\rightarrow$ TERM*}

Hiddens

Context-free Functions

ord $\rightarrow$ Combinator {**type:** RULE $\rightarrow$ n}

Equations

The combinator *rule* constructs a rewrite rule from a *kind* (*norm* for normal rules and *def* for default rules), a *tag*, a list of conditions and two terms (the lhs and rhs).

$$\mathit{rule}\ \mathit{norm}\ f\ [\![]\!]_{\vec{C}}\ t_1\ t_2\ =\ [f]\ t_1 = t_2$$

$$\mathit{rule}\ \mathit{norm}\ f\ [\![C, \vec{C}]\!]_{\vec{C}}\ t_1\ t_2\ =\ [f]\ C, \vec{C} \texttt{===>}\ t_1 = t_2$$

$$\begin{aligned}&\mathit{rule}\ \mathit{def}\ f\ [\![\vec{C}]\!]_{\vec{C}}\ t_1\ t_2 = \mathit{otherwise} : \varphi\\ &\qquad\qquad\qquad\qquad \textbf{when}\\ &\qquad\qquad\qquad\qquad\quad \varphi = \mathit{rule}\ \mathit{norm}\ f\ [\![\vec{C}]\!]_{\vec{C}}\ t_1\ t_2\end{aligned}$$

The components of a rule are extracted by the combinators *kind*, *tag*, *conds*, *lhs* and *rhs*.

$$\begin{array}{llll} \mathit{kind}\ [f]\ s = t & = \mathit{norm} & \mathit{tag}\ [f]\ s = t & = f \\ \mathit{kind}\ [f]\ C^+ \Rightarrow s = t & = \mathit{norm} & \mathit{tag}\ [f]\ C^+ \Rightarrow s = t & = f \\ \mathit{kind}\ \mathit{otherwise} : \varphi & = \mathit{def} & \mathit{tag}\ \mathit{otherwise} : \varphi & = \mathit{tag}\ \varphi \end{array}$$

$$\begin{array}{ll} \mathit{conds}\ [f]\ s = t & = [\![]\!]_{\vec{C}} \\ \mathit{conds}\ [f]\ C^+ \Rightarrow s = t & = [\![C^+]\!]_{\vec{C}} \\ \mathit{conds}\ \mathit{otherwise} : \varphi & = \mathit{conds}\ \varphi \end{array}$$

$$\begin{array}{llll} \mathit{lhs}\ [f]\ s = t & = s & \mathit{rhs}\ [f]\ s = t & = t \\ \mathit{lhs}\ [f]\ C^+ \Rightarrow s = t & = s & \mathit{rhs}\ [f]\ C^+ \Rightarrow s = t & = t \\ \mathit{lhs}\ \mathit{otherwise} : \varphi & = \mathit{lhs}\ \varphi & \mathit{rhs}\ \mathit{otherwise} : \varphi & = \mathit{rhs}\ \varphi \end{array}$$

We say that *rule* φ *defines function* f, if $\mathit{fun}\ \varphi = f$, where *fun* is defined as

$$\mathit{fun}\ \varphi = \mathit{fun}\ (\mathit{lhs}\ \varphi)$$

The variables of a rule are

$$\mathit{vars}\ \varphi = \mathit{vars}\ (\mathit{lhs}\ \varphi) +\!\!+ \downarrow_{\vec{t}} (\mathit{flatMap}\ (\mathit{vars} \circ \mathit{lhs})\ (\mathit{filter}\ \mathit{new?}\ (\mathit{conds}\ \varphi)))$$

The terms of a rule are

$$\mathit{terms}\ \varphi = \mathit{lhs}\ \varphi : \mathit{flatMap}\ (\lambda\ \_C\ .\ \mathit{lhs}\ \_C : \mathit{rhs}\ \_C : [\![]\!]_{\vec{t}})\ (\mathit{conds}\ \varphi) +\!\!+ \mathit{rhs}\ \varphi : [\![]\!]_{\vec{t}}$$

Rule Priority The relation $\leq$ on rules dictates the order in which rules should be tried in rewriting. Rules may be compared if they define the same function.

$$\varphi_1 \leq \varphi_2 = \mathit{ord}\ \varphi_1 \leq \mathit{ord}\ \varphi_2 \Leftarrow \mathit{fun}\ \varphi_1 = \mathit{fun}\ \varphi_2$$

where *ord* is defined as

$$\begin{aligned}
&\mathit{ord}\ [f]\ t_1 = t_2 &&= 0\\
&\mathit{ord}\ [f]\ C^+ \Rightarrow t_1 = t_2 &&= 0\\
&\mathit{ord}\ \mathit{otherwise} : \varphi &&= 1 + \mathit{ord}\ \varphi
\end{aligned}$$

This means that default rules have a higher *ord* than normal rules. A higher *ord* entails a lower priority; default rules are tried after all normal rules have been tried. The relation $>$ is the complement of $\leq$.

$$\varphi_1 > \varphi_2 = \neg\ (\varphi_2 \leq \varphi_1)$$

9.4.7 TRS

Imports Rewrite-Rules(9.4.6)

We define combinators for construction, decompositions and union of TRSs.

Exports

Context-free Functions

trs $\rightarrow$ Combinator {**type:** FUNDECL* $\rightarrow$ RULE* $\rightarrow$ TRS}
"Σ" $\rightarrow$ Combinator {**type:** TRS $\rightarrow$ FUNDECL*}
"$\mathbf{R}$" $\rightarrow$ Combinator {**type:** TRS $\rightarrow$ RULE*}

Equations

The combinator *trs* constructs a TRS from a list of function declarations and a list of rules.

$$\mathit{trs}\ [\![\vec{\mathit{fd}}\ ]\!]_{\vec{\mathit{fd}}}\ [\![\vec{\varphi}]\!]_{\vec{\varphi}} = \mathit{signature}\ \vec{\mathit{fd}}\ \mathit{equations}\ \vec{\varphi}$$

The combinator Σ yields the function declarations or signature of a TRS. The combinator $\mathbf{R}$ yields the rules or equations of a TRS.

$$\begin{aligned}
&\Sigma\ \mathit{signature}\ \vec{\mathit{fd}}\ \mathit{equations}\ \vec{\varphi} = [\![\vec{\mathit{fd}}\ ]\!]_{\vec{\mathit{fd}}}\\
&\mathbf{R}\ \mathit{signature}\ \vec{\mathit{fd}}\ \mathit{equations}\ \vec{\varphi} = [\![\vec{\varphi}]\!]_{\vec{\varphi}}
\end{aligned}$$

Union of specifications.

$$\mathcal{T}_1 + \mathcal{T}_2 = \mathit{trs}\ (\Sigma\ \mathcal{T}_1 +\!\!+\ \Sigma\ \mathcal{T}_2)\ (\mathbf{R}\ \mathcal{T}_1 +\!\!+\ \mathbf{R}\ \mathcal{T}_2)$$

Note that the rules of $\mathcal{T}_1$ come before the rules of $\mathcal{T}_2$ and have therefore higher priority in rewriting.

9.5 Environments

Imports RN2-Terminology(9.4) Assignments(11.1)

We define the datastructure *environments*. It is used in transformations to store global information. Projection, retrieval and update functions are defined for each field.

The type $\mathbb{E}$ will be used in type annotations to denote environments. It is defined as

$$\begin{aligned}\mathbb{E} = \; & \text{FUNDECL}^* \times \text{RULE}^* \times \text{CONDITION}^* \times \text{TERM}^* \times n^* \\ & \times \text{TERM} \times \text{TERM} \times \text{TERM} \times \text{TERM} \times \text{TERM} \times \text{TERM}^* \\ & \times \text{Assign} \times \text{FUNSYM}^* \times \text{APVAR}^* \times \mathbb{B}\end{aligned}$$

Exports

Context-free Functions

new-env $\rightarrow$ Combinator {**type:** FUNSYM $\rightarrow$ APVAR $\rightarrow \mathbb{E}$}
new-env-def $\rightarrow$ Combinator {**type:** $\mathbb{E}$}

Variables

"$\mathcal{E}$" [*0-9'*]$*$ $\rightarrow$ Term
"Θ^v" [*0-9'*]$*$ $\rightarrow$ Term

The variable $\mathcal{E}$ will be used to denote environments.

Exports

Context-free Functions

"$\pi_{\vec{fd}}$" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ FUNDECL*}
"$\pi_{\vec{\varphi}}$" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ RULE*}
"$\pi_{\vec{C}}$" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ COND*}
"π_{vars}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM*}
"π_{last}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow \mathbb{N}^*$}
"π_{lhs}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM}
"π_{rhs}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM}
"π_{clhs}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM}
"π_{crhs}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM}
"π_t" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM}
"$\pi_{\vec{t}}$" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM*}
"π_ρ" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ (VAR $\rightarrow$ TERM)}
"π_{nvars}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM*}
"π_{nfuns}" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ FUN*}
"$\pi_{\mathcal{B}}$" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow \mathbb{B}$}

Exports

Context-free Functions

"$set_{\vec{fd}}$" $\rightarrow$ Combinator {**type:** FUNDECL* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"$set_{\vec{\varphi}}$" $\rightarrow$ Combinator {**type:** RULE* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"$set_{\vec{C}}$" $\rightarrow$ Combinator {**type:** COND* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{vars}" $\rightarrow$ Combinator {**type:** TERM* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{last}" $\rightarrow$ Combinator {**type:** $\mathbb{N}^* \rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{lhs}" $\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{rhs}" $\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{clhs}" $\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{crhs}" $\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_t" $\rightarrow$ Combinator {**type:** TERM $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"$set_{\vec{t}}$" $\rightarrow$ Combinator {**type:** TERM* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_ρ" $\rightarrow$ Combinator {**type:** (VAR $\rightarrow$ TERM) $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{nvars}" $\rightarrow$ Combinator {**type:** VAR* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"set_{nfuns}" $\rightarrow$ Combinator {**type:** FUN* $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"$set_{\mathcal{B}}$" $\rightarrow$ Combinator {**type:** $\mathbb{B} \rightarrow \mathbb{E} \rightarrow \mathbb{E}$}

Exports

Context-free Functions

"$+\!\!+_{\vec{fd}}$" $\rightarrow$ Infix-Operator {**type:** FUNDECl* $\times \mathbb{E} \rightarrow \mathbb{E}$}
"$+\!\!+_{\vec{\varphi}}$" $\rightarrow$ Infix-Operator {**type:** RULE* $\times \mathbb{E} \rightarrow \mathbb{E}$}
"$+\!\!+_{\vec{C}}$" $\rightarrow$ Infix-Operator {**type:** COND* $\times \mathbb{E} \rightarrow \mathbb{E}$}

"$+\!\!+_{vars}$" $\rightarrow$ Infix-Operator {**type:** $\text{TERM}^* \times \mathbb{E} \rightarrow \mathbb{E}$}
"$:_{last}$" $\rightarrow$ Infix-Operator {**type:** $\mathbb{N}^* \times \mathbb{E} \rightarrow \mathbb{E}$}
"$+\!\!+_{\vec{t}}$" $\rightarrow$ Infix-Operator {**type:** $\text{TERM}^* \times \mathbb{E} \rightarrow \mathbb{E}$}
assign $\rightarrow$ Combinator {**type:** $(\text{VAR} \rightarrow \text{TERM}) \times \mathbb{E} \rightarrow \mathbb{E}$}
"$\wedge_{\mathcal{B}}$" $\rightarrow$ Infix-Operator {**type:** $\mathbb{B} \times \mathbb{E} \rightarrow \mathbb{E}$}

Exports

Context-free Functions

"$:_{fd}$" $\rightarrow$ Infix-Operator {**type:** $\text{FUNDECl} \times \mathbb{E} \rightarrow \mathbb{E}$}
"$:_{\varphi}$" $\rightarrow$ Infix-Operator {**type:** $\text{RULE} \times \mathbb{E} \rightarrow \mathbb{E}$}
"$:_{C}$" $\rightarrow$ Infix-Operator {**type:** $\text{COND} \times \mathbb{E} \rightarrow \mathbb{E}$}
"$:_{x^{(*)}}$" $\rightarrow$ Infix-Operator {**type:** $\text{TERM} \times \mathbb{E} \rightarrow \mathbb{E}$}
"$:_{t}$" $\rightarrow$ Infix-Operator {**type:** $\text{TERM} \times \mathbb{E} \rightarrow \mathbb{E}$}

Exports

Context-free Functions

alloc-fun $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow (\mathbb{E} \times \text{FUNSYM})$}
alloc-var $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow (\mathbb{E} \times \text{APVAR})$}
alloc-slvar $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow (\mathbb{E} \times \text{SLVAR})$}
alloc-nfuns $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow (\mathbb{E} \times \text{FUNSYM}^*)$}
alloc-nvars $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow (\mathbb{E} \times \text{VAR}^*)$}

Exports

Context-free Functions

newVars $\rightarrow$ Combinator {**type:** $\text{VAR} \rightarrow \text{VAR}^*$}
newVarsi $\rightarrow$ Combinator {**type:** $\mathbb{N} \rightarrow \text{VAR} \rightarrow \text{VAR}^*$}
newFuns $\rightarrow$ Combinator {**type:** $\text{FUNSYM} \rightarrow \text{FUNSYM}^*$}
newFunsi $\rightarrow$ Combinator {**type:** $\mathbb{N} \rightarrow \text{FUNSYM} \rightarrow \text{FUNSYM}^*$}

Hiddens

Variables

[*X*] [*0-9'*]$*$ $\rightarrow$ Term

Equations

New-env constructs a new environment from a function symbol and a normal variable that are used to initialize the supplies of functions and variables. The list fields are initialized with empty lists.

$$\begin{aligned} \textit{new-env}\ f\ x\ =\ \langle & [\![\ ]\!]_{\vec{fd}}, [\![]\!]_{\vec{\varphi}}, [\![]\!]_{\vec{C}}, [\![]\!]_{\vec{t}}, [], Z, Z, Z, Z, Z, [\![]\!]_{\vec{t}}, \\ & \Delta_\rho\ \mathbf{I},\ \textit{newFuns}\ f,\ \textit{newVars}\ x,\ \top \rangle \end{aligned}$$

When we do not care about the variable and function prefix used, *new-env-def* constructs a default environment.

$$\textit{new-env-def} = \textit{new-env}\ \$\textit{def}\ \text{DEF}$$

The projection functions (with prefix π) and update functions (with prefix *set*) are synonyms for π and *set* with a fixed index.

$$\begin{array}{llll}
\pi_{\vec{fd}} = \pi\ 1 & set_{\vec{fd}} = set\ 1 & \pi_{\vec{\varphi}} = \pi\ 2 & set_{\vec{\varphi}} = set\ 2 \\
\pi_{\vec{C}} = \pi\ 3 & set_{\vec{C}} = set\ 3 & \pi_{vars} = \pi\ 4 & set_{vars} = set\ 4 \\
\pi_{last} = \pi\ 5 & set_{last} = set\ 5 & \pi_{lhs} = \pi\ 6 & set_{lhs} = set\ 6 \\
\pi_{rhs} = \pi\ 7 & set_{rhs} = set\ 7 & \pi_{clhs} = \pi\ 8 & set_{clhs} = set\ 8 \\
\pi_{crhs} = \pi\ 9 & set_{crhs} = set\ 9 & \pi_{t} = \pi\ 10 & set_{t} = set\ 10 \\
\pi_{\vec{t}} = \pi\ 11 & set_{\vec{t}} = set\ 11 & \pi_{\rho} = \pi\ 12 & set_{\rho} = set\ 12 \\
\pi_{nfuns} = \pi\ 13 & set_{nfuns} = set\ 13 & \pi_{nvars} = \pi\ 14 & set_{nvars} = set\ 14 \\
\pi_{\mathcal{B}} = \pi\ 15 & set_{\mathcal{B}} = set\ 15 & &
\end{array}$$

The ++-functions concatenate a list to the corresponding list in the environment.

$$
\begin{aligned}
[\![\vec{fd}\,]\!]_{\vec{fd}} +\!\!+_{\vec{fd}}\, \mathcal{E} &= set_{\vec{fd}}\ ([\![\vec{fd}\,]\!]_{\vec{fd}} +\!\!+ \pi_{\vec{fd}}\ \mathcal{E})\ \mathcal{E}\\
[\![\vec{\varphi}]\!]_{\vec{\varphi}} +\!\!+_{\vec{\varphi}}\, \mathcal{E} &= set_{\vec{\varphi}}\ ([\![\vec{\varphi}]\!]_{\vec{\varphi}} +\!\!+ \pi_{\vec{\varphi}}\ \mathcal{E})\ \mathcal{E}\\
[\![\vec{C}]\!]_{\vec{C}} +\!\!+_{\vec{C}}\, \mathcal{E} &= set_{\vec{C}}\ ([\![\vec{C}]\!]_{\vec{C}} +\!\!+ \pi_{\vec{C}}\ \mathcal{E})\ \mathcal{E}\\
\Theta^v +\!\!+_{vars}\, \mathcal{E} &= set_{vars}\ (\Theta^v +\!\!+ \pi_{vars}\ \mathcal{E})\ \mathcal{E}\\
n :_{last} \mathcal{E} &= set_{last}\ (n : \pi_{last}\ \mathcal{E})\ \mathcal{E}\\
[\![\vec{t}]\!]_{\vec{t}} +\!\!+_{\vec{t}}\, \mathcal{E} &= set_{\vec{t}}\ ([\![\vec{t}]\!]_{\vec{t}} +\!\!+ \pi_{\vec{t}}\ \mathcal{E})\ \mathcal{E}\\
assign\ \rho\ \mathcal{E} &= set_{\rho}\ (\rho \oplus \pi_{\rho}\ \mathcal{E})\ \mathcal{E}\\
\mathcal{B} \wedge_{\mathcal{B}} \mathcal{E} &= set_{\mathcal{B}}\ (\mathcal{B} \wedge \pi_{\mathcal{B}}\ \mathcal{E})\ \mathcal{E}
\end{aligned}
$$

The :-functions add one item to the corresponding list in the environment.

$$
\begin{aligned}
fd :_{fd} \mathcal{E} &= set_{\vec{fd}}\ (fd : \pi_{\vec{fd}}\ \mathcal{E})\ \mathcal{E}\\
\varphi :_{\varphi} \mathcal{E} &= set_{\vec{\varphi}}\ (\varphi : \pi_{\vec{\varphi}}\ \mathcal{E})\ \mathcal{E}\\
C :_{C} \mathcal{E} &= set_{\vec{C}}\ (C : \pi_{\vec{C}}\ \mathcal{E})\ \mathcal{E}\\
x^{(*)} :_{x^{(*)}} \mathcal{E} &= set_{vars}\ (x^{(*)} : \pi_{vars}\ \mathcal{E})\ \mathcal{E}\\
t :_{t} \mathcal{E} &= set_{\vec{t}}\ (t : \pi_{\vec{t}}\ \mathcal{E})\ \mathcal{E}
\end{aligned}
$$

Allocation of new functions and variables. The *nfuns* and *nvars* fields contain a list of new functions and a list of new variables. The *alloc* functions allocate a new function or variable and update the list in the environment.

$$
\begin{aligned}
alloc\text{-}fun\ \mathcal{E} &= \langle set_{nfuns}\ (tl\ X)\ \mathcal{E},\ hd\ X\rangle \Leftarrow X = \pi_{nfuns}\ \mathcal{E}\\
alloc\text{-}var\ \mathcal{E} &= \langle set_{nvars}\ (tl\ X)\ \mathcal{E},\ hd\ X\rangle \Leftarrow X = \pi_{nvars}\ \mathcal{E}\\
alloc\text{-}slvar\ \mathcal{E} &= \langle \mathcal{E}',\ x\ *\rangle \qquad\qquad \Leftarrow \langle \mathcal{E}',\ x\rangle = alloc\text{-}var\ \mathcal{E}
\end{aligned}
$$

$$
\begin{aligned}
alloc\text{-}nfuns\ n\ \mathcal{E} &= \langle set_{nfuns}\ (drop\ n\ X)\ \mathcal{E},\ take\ n\ X\rangle \Leftarrow X = \pi_{nfuns}\ \mathcal{E}\\
alloc\text{-}nvars\ n\ \mathcal{E} &= \langle set_{nvars}\ (drop\ n\ X)\ \mathcal{E},\ take\ n\ X\rangle \Leftarrow X = \pi_{nvars}\ \mathcal{E}
\end{aligned}
$$

Supplies of Variables and Functionsymbols *NewVars* generates an infinite list of variables.

$$newVars\ x^{(*)} = newVarsi\ 0\ x^{(*)}$$

The infinite list of variables is constructed by *newVarsi*. The primitive list operations are defined such that the list never ends.

$$
\begin{aligned}
\varepsilon?\ (newVarsi\ n\ x^{(*)}) &= \bot\\
hd\ (newVarsi\ n\ x^{(*)}) &= x^{(*)}\ \_\ n\\
tl\ (newVarsi\ n\ x^{(*)}) &= newVarsi\ (n+1)\ x^{(*)}\\
nil\ (newVarsi\ n\ x^{(*)}) &= [\![]\!]_{\vec{t}}
\end{aligned}
$$

NewFuns generates an infinite supply of function names.

$$newFuns\ f = newFunsi\ 0\ f$$

newFunsi is analogous to *newVarsi*.

$$
\begin{aligned}
\varepsilon?\ (newFunsi\ n\ f) &= \bot\\
hd\ (newFunsi\ n\ f) &= f\ \_\ n\\
tl\ (newFunsi\ n\ f) &= newFunsi\ (n+1)\ f\\
nil\ (newFunsi\ n\ f) &= []
\end{aligned}
$$

Chapter 10

Static Semantics

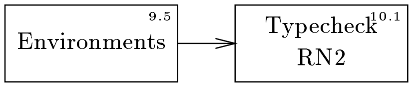


Figure 10.1: Import graph for Chapter 10; Static Semantics

10.1 Typecheck RN2

Imports Environments$^{(9.5)}$

We define the (first-order) functions that check the static correctness of TRSs and of TERMs as terms over the signature of a TRS. These functions apply to TRSs and TERMs in the language RN2 and any of its sublanguages.

Exports

Context-free Functions

"*tc*" "⟦" TRS "⟧"	$\to$ Boolean
"*tc*" "⟦" TRS "," TERM "⟧"	$\to$ Boolean
"$tc_{\mathcal{T}}$"	$\to$ Combinator {**type:** TRS $\to \mathbb{B}$}

The rules for static correctness of a TRS are

1. Functions may be declared only once in the signature.
2. The lhs of a rule should be an application.
3. Sublist variables may occur only as direct subterms of lists.
4. Variables in a matching term may not occur in previous terms.
5. Variables in instantiating terms should have occurred in a previous matching term.
6. The list of types of the arguments of an application of a function f should be a subtype of the list of types of the declaration of f in in the signature.

A TERM is correct in the context of a TRS if

1. Applications are applications of a function declared in the signature of the TRS, with the right number of arguments. The types of the arguments should be subtypes of the corresponding types of the function in the signature.
2. The term has no variables; it is a ground term.

We refer to these rules in the explanation of the specification by their numbers.

The functions are specified by means of the following combinators.

Hiddens

Context-free Functions

"tc_{fd}" $\rightarrow$ Combinator {**type:** FUNDECL $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"tc_{φ}" $\rightarrow$ Combinator {**type:** RULE $\rightarrow \mathbb{E} \rightarrow \mathbb{E}$}
"tc_C" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ COND $\rightarrow \mathbb{E}$}
"tc_t^r" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ TERM $\rightarrow \mathbb{E} \times$ TYPE}
"tc_t" $\rightarrow$ Combinator {**type:** $\mathbb{E} \rightarrow$ (VAR $\cup$ FUNSYM $\cup$ {list}) $\rightarrow$ TYPE$*$ $\rightarrow \mathbb{E} \times$ TYPE}

Equations

The static correctness of a TRS is implemented by combinator $tc_{\mathcal{T}}$.

$$tc[\![\mathcal{T}]\!] = \mathcal{B} \Leftarrow \mathcal{B} = tc_{\mathcal{T}}\ \mathcal{T}$$

A TERM is statically correct as a term over the signature of some TRS if all its functions appear in the signature with the right types and if it is a ground term.

$$\frac{\mathcal{B} = \pi_{\mathcal{B}}\ (\pi\ 1\ (tc_t^r\ (set_{\vec{fd}}\ (\Sigma\ \mathcal{T})\ \textit{new-env-def})\ t)) \wedge \varepsilon?\ (vars\ t)}{tc[\![\mathcal{T}, t]\!] = \mathcal{B}}$$

TRSs The $\mathcal{B}$ field of the environment is used to keep track of the result of type-checking. The field is initialized to $\top$. The rules are checked after checking the function declarations.

$$tc_{\mathcal{T}}\ \mathcal{T} = \pi_{\mathcal{B}}\ (foldr\ tc_{\varphi}\ (foldr\ tc_{fd}\ (set_{\mathcal{B}}\ \top\ \textit{new-env-def})\ (\Sigma\ \mathcal{T}))\ (\mathbf{R}\ \mathcal{T}))$$

Function Declarations A function declaration is correct if the function was not declared already (1). The function declaration is added to the environment.

$$tc_{fd}\ f(\vec{\tau})\ \mathcal{E}\ =\ \neg\ (\pi_{\vec{fd}}\ \mathcal{E}\ ?\ f) \wedge_{\mathcal{B}} (f(\vec{\tau}) :_{fd} \mathcal{E})$$

Rules are correct if the lhs is an application (2), the rhs is not a sublist variable (3), the variables of the rhs are old variables (5), if the lhs and rhs are type correct, and if the conditions are correct.

$$\begin{array}{l} tc_{\varphi}\ \varphi\ \mathcal{E}_1 = \tau_1 \equiv appTp \wedge (\tau_2 \preceq lstTp) \wedge (vars\ (rhs\ \varphi) \subseteq \pi_{vars}\ \mathcal{E}_4) \wedge_{\mathcal{B}} \mathcal{E}_4 \\ \qquad \textbf{when} \\ \qquad \langle \mathcal{E}_2, \tau_1 \rangle = tc_t^r\ \mathcal{E}_1\ (lhs\ \varphi), \\ \qquad \mathcal{E}_3 \qquad = foldl\ tc_C\ (set_{vars}\ (vars\ (lhs\ \varphi))\ \mathcal{E}_2)\ (conds\ \varphi), \\ \qquad \langle \mathcal{E}_4, \tau_2 \rangle = tc_t^r\ \mathcal{E}_3\ (rhs\ \varphi) \end{array}$$

Conditions A positive or negative condition is correct if lhs and rhs are correct, if they are not sublist variables (3) and if the variables of the lhs and rhs are old variables (5).

$$\frac{kind\ C \neq new,\ \langle \mathcal{E}_2, \tau_1 \rangle = tc_t^r\ \mathcal{E}_1\ (rhs\ C),\ \langle \mathcal{E}_3, \tau_2 \rangle = tc_t^r\ \mathcal{E}_2\ (lhs\ C)}{\begin{array}{l} tc_C\ \mathcal{E}_1\ C = \tau_1 \preceq lstTp \wedge (\tau_2 \preceq lstTp) \\ \qquad \wedge\ (vars\ (lhs\ C) \subseteq \pi_{vars}\ \mathcal{E}_3) \wedge (vars\ (rhs\ C) \subseteq \pi_{vars}\ \mathcal{E}_3) \wedge_{\mathcal{B}} \mathcal{E}_3 \end{array}}$$

For an assignment condition the same requirements hold, except for the variables of the lhs (a matching term). The variables of the lhs should be new variables (4).

These variables are added to the list of variables in the environment.

$$\frac{\langle \mathcal{E}_2, \tau_1\rangle = tc_t^r\ \mathcal{E}_1\ s,\ \ \langle \mathcal{E}_3, \tau_2\rangle = tc_t^r\ \mathcal{E}_2\ t,\ \ [\![\vec{a}]\!]_{\vec{t}} = vars\ t}{\begin{aligned} tc_C\ \mathcal{E}_1\ t := s = {} & \tau_1 \preceq lstTp \wedge (\tau_2 \preceq lstTp) \\ & \wedge (vars\ s \subseteq \pi_{vars}\ \mathcal{E}_3) \wedge \neg\ ([\![\vec{a}]\!]_{\vec{t}} \subseteq \pi_{vars}\ \mathcal{E}_3) \\ & \wedge_{\mathcal{B}}\ ([\![\vec{a}]\!]_{\vec{t}} +\!\!+_{vars}\ \mathcal{E}_3) \end{aligned}}$$

Terms tc_t^r recursively checks if all subterms of a term are correct with respect to rules (3) and (6) and returns the updated environment and the type of the term. The recursion is handled by *poAcc*, and the cases are dealt with by tc_t.

$$tc_t^r\ \mathcal{E}\ t = poAcc\ tc_t\ [\![]\!]_{\vec{\tau}}\ \mathcal{E}\ t$$

A variable is always correct.

$$tc_t\ \mathcal{E}\ x^{(*)}\ [\![]\!]_{\vec{\tau}} = \langle \mathcal{E},\ type\ x^{(*)}\rangle$$

A list is correct if all its arguments have a type that is a subtype of *slTp*. This means that they may have any type.

$$tc_t\ \mathcal{E}\ list\ [\![\vec{\tau}]\!]_{\vec{\tau}} = \langle \forall\ (\preceq slTp)\ [\![\vec{\tau}]\!]_{\vec{\tau}} \wedge_{\mathcal{B}} \mathcal{E},\ lstTp\rangle$$

An application of a function f to subterms with types $\vec{\tau}$ is correct if f is declared in the signature as $f(\vec{\tau}')$ and if $[\![\vec{\tau}]\!]_{\vec{\tau}}$ is a subtype of $[\![\vec{\tau}']\!]_{\vec{\tau}}$.

$$tc_t\ \mathcal{E}\ f\ [\![\vec{\tau}]\!]_{\vec{\tau}}\ =\ \langle \pi_{\vec{fd}}\ \mathcal{E}\ ?\ f \wedge ([\![\vec{\tau}]\!]_{\vec{\tau}} \preceq (\pi_{\vec{fd}}\ \mathcal{E}\ .\ f)) \wedge_{\mathcal{B}} \mathcal{E},\ appTp\rangle$$

Chapter 11

Dynamic Semantics

We give a specification of the dynamic semantics of RN2. The specification reflects the execution of ASF+SDF specifications as TRSs as described in [Wal91]. The differences with that description are

- All possible matches for a term and a pattern are computed at once. In a real implementation a new match is only sought if the previous one did not lead to success.
- The rules for a function are tried in the order in which they appear in the specification. For confluent TRSs this makes no difference.
- Rules are partitioned in normal and default rules. A default rule for a function is tried after all normal rules for that function are tried. More than one default rule for a function may exist. In that case first all normal rules are tried. If none of the normal rules apply the default rules are tried in the order in which they appear in the specification.

  Actually, in the compiler, only one default rule for a function may be used. Allowing more than one default rule, as we do here, does not entail a different semantics for TRSs which have at most one default rule per function.

We specify the dynamic semantics to have a tool to check the correctness of transformations on TRSs.

For the specification of term reduction we need some preliminaries. In Section 11.1 a definition of assignments is given. In Section 11.2 the substitution of variables in terms given some assignment is specified. Matching is defined in Section 11.3. Section 11.4 defines a function that translates a list of rules to a table that maps a function symbol to the list of rules which define it. Finally, in Section 11.5, the definition of reduction is given.

The import graph for this chapter is shown in Figure 11.1.

11.1 Assignments

Imports Tools$^{(2.2.1)}$

In the next sections assignments will be used as functions that map APVARs to TERMs and SLVARs to lists of TERMs. Since assignments may have a broader application than this we define assignments in a more general way.

An assignment is a function that maps Constants to Constants. (Sort Constant is abbreviated as $\mathbb{C}$ in types). Each assignment is a default function with a table of exceptions.

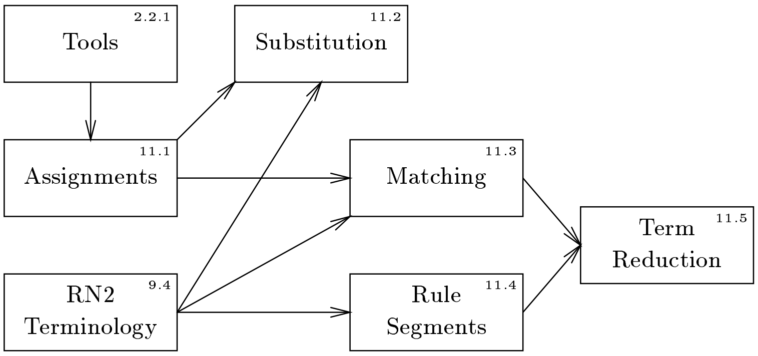


Figure 11.1: Import graph for Chapter 11; Dynamic Semantics

The following functions are provided for constructing assignments.

Exports

Context-free Functions

"$\mapsto$"	$\rightarrow$ Infix-Operator	{**type:** $(\mathbb{C} \times \mathbb{C}) \rightarrow \mathbb{C} \rightarrow \mathbb{C}$}
"$\oplus$"	$\rightarrow$ Right-Infix-Operator	{**type:** $(\mathbb{C} \rightarrow \mathbb{C}) \times (\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{C} \rightarrow \mathbb{C}$}
"$\boxplus$"	$\rightarrow$ Right-Infix-Operator	{**type:** $(\mathbb{C} \rightarrow \mathbb{C}) \times (\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{C} \rightarrow \mathbb{C}$}
"$\perp_\rho$"	$\rightarrow$ Combinator	{**type:** $\mathbb{C} \rightarrow \mathbb{C}$}
"Δ_ρ"	$\rightarrow$ Combinator	{**type:** $(\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{C} \rightarrow \mathbb{C}$}
domain	$\rightarrow$ Combinator	{**type:** $(\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{C}^*$}
range	$\rightarrow$ Combinator	{**type:** $(\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{C}^*$}
undefine	$\rightarrow$ Combinator	{**type:** $\mathbb{C} \rightarrow (\mathbb{C} \rightarrow \mathbb{C}) \rightarrow (\mathbb{C} \rightarrow \mathbb{C})$}
"*defines?*"	$\rightarrow$ Combinator	{**type:** $\mathbb{C} \rightarrow (\mathbb{C} \rightarrow \mathbb{C}) \rightarrow \mathbb{B}$}

Variables

"ρ" [*0-9'*]* $\rightarrow$ Term

"$\vec{\rho}$" [*0-9'*]* $\rightarrow$ {Term "," }*

We illustrate the usage of assignments with the following example.

Example (Implementation of sqr) Programmer Y gets the specification $sqr\ x = x^2$. The function is incrementally defined using assignments. The first prototype is

$$\rho_1 = (1 \mapsto 1) \oplus (2 \mapsto 4) \oplus (3 \mapsto 9)$$

This seems to work properly; $\rho_1\ 2$ yields 4. Then error handling is added. On incorrect input the function responds with an error message.

$$\rho_2 = \rho_1 \oplus (\Delta_\rho((\text{“error: sqr not defined for: ”}\ +\!\!+) \circ \downarrow_{\text{“”}}))$$

For instance, on input "abc", ρ_2 yields

"error: sqr not defined for: abc"

Beta testing then reveals a bug; $\rho_2\ 4$ does not yield the expected value. The bug is fixed in version 3.

$$\rho_3 = (4 \mapsto 12) \oplus \rho_2$$

Now $\rho_3\ 4$ yields 12. Users complain about this and another bug fix results in version 4. The old value of 4 is overruled.

$$\rho_4 = (4 \mapsto 16) \boxplus \rho_3$$

Now the function seems completely defined and may be implemented by the following Pascal function.

```
function sqr(x:integer):integer; begin
    case x of
        1: sqr := 1; 2: sqr := 4; 3: sqr := 9; 4: sqr := 16;
        otherwise
          writeln('error: sqr undefined for: ',  x)
    end
end
```

Internally all assignments are represented by an application of *asgn* to a table and a default function.

Hiddens

Context-free Functions

$asgn \rightarrow$ Combinator

$add \rightarrow$ Combinator

Variables

$[FL]$ $[0\text{-}9']*$ $\rightarrow$ Term

Equations

Auxiliary Functions First we define the hidden function *asgn*. It has two arguments. The first is a table of pairs of constants and the second a default function. This yields a function that maps constants to constants as follows.

$$asgn\ L\ F\ \delta = \boldsymbol{if}^c\ \langle L\ ?\ \delta,\ L\ .\ \delta,\ F\ \delta\rangle$$

That is, if there is an entry for δ in the table, the value of the entry is returned else the default function is applied to δ.

Next we define the hidden function *add*, which adds a new entry $\langle\delta_1, \delta_2\rangle$ to the exception table of an assignment. There are three cases. If there was no entry for δ_1 the entry is added to the table. If there was an entry with the same value the assignment does not change. If there was an entry for δ_1 with an other value than δ_2 adding the entry fails ($\perp_\rho$).

$$\begin{aligned}
add\ \langle\delta_1,\ \delta_2\rangle\ (asgn\ L\ F) &= asgn\ (\langle\delta_1,\ \delta_2\rangle : L)\ F \Leftarrow L\ ?\ \delta_1 = \perp \\
add\ \langle\delta_1,\ \delta_2\rangle\ (asgn\ L\ F) &= asgn\ L\ F \qquad\qquad \Leftarrow L\ ?\ \delta_1 = \top,\ \ L\ .\ \delta_1 \equiv \delta_2 = \top \\
add\ \langle\delta_1,\ \delta_2\rangle\ (asgn\ L\ F) &= \perp_\rho \qquad\qquad\qquad \Leftarrow L\ ?\ \delta_1 = \top,\ \ L\ .\ \delta_1 \equiv \delta_2 = \perp
\end{aligned}$$

Application of a failure assignment ($\perp_\rho$) to a constant is undefined.

Constructors An atomic assignment is a single exception to the identity function.

$$\delta_1 \mapsto \delta_2 = asgn\ [\langle\delta_1,\ \delta_2\rangle]\ \mathbf{I}$$

Any function with type ($\mathbb{C} \rightarrow \mathbb{C}$) is converted to an assignment by Δ_ρ.

$$\Delta_\rho\ F = asgn\ []\ F$$

Assignments are composed by $\oplus$. The default of the second assignment is kept as the default of the new assignment. No overlap between the two exception tables may exist.

$$asgn\ L_1\ F_1 \oplus asgn\ L_2\ F_2 = foldr\ add\ (asgn\ L_2\ F_2)\ L_1$$

An exception is removed from an assignment by *undefine*.

$$undefine\ \delta\ (asgn\ L\ F) = asgn\ (rm\ \delta\ L)\ F$$

The exceptions and the default function of an assignment may be overruled by $\boxplus$. All exceptions in the second assignment that are not defined in the first are kept in the new assignment.

$$\begin{aligned} \mathit{asgn}\ L_1\ F_1 \boxplus \mathit{asgn}\ L_2\ F_2 \ &= \ \mathit{asgn}\ L_1\ F_1 \\ &\quad \oplus \mathit{foldr}\ \mathit{undefine}\ (\mathit{asgn}\ L_2\ F_1)\ (\mathit{map}\ \mathit{key}\ L_1) \end{aligned}$$

The *domain* and *range* of the exceptions in an assignment.

$$\mathit{domain}\ (\mathit{asgn}\ L\ F) = \mathit{map}\ \mathit{key}\ L \qquad \mathit{range}\ (\mathit{asgn}\ L\ F) = \mathit{map}\ \mathit{value}\ L$$

An assignment *defines?* a constant δ if there is entry for δ in the exception table of the assignment.

$$\mathit{defines?}\ \delta\ (\mathit{asgn}\ L\ F) = L\ ?\ \delta$$

11.2 Substitution

Imports RN2-Terminology$^{(9.4)}$ Assignments$^{(11.1)}$

Given an assignment ρ that maps APVARs to TERMs and SLVARs to lists of TERMs, *subst* replaces all variables in a term by their values in ρ. The function *substp* is a more general substitution operator that is parameterized with a function that knows how to handle lists and a representation for the empty list. This is useful when we want to convert associative lists in terms to lists constructed by *cons* and *nil*.

We will abbreviate the type of assignments used by *subst* as

$$\text{Assign} = (\text{APVAR} \to \text{TERM}) \cup (\text{SLVAR} \to \text{TERM}^*)$$

Exports

Context-free Functions

substp $\to$ Combinator {**type:** $((\text{APVAR} \times \text{TERM}) \cup (\text{SLVAR} \times \beta) \to \text{TERM})$
$\to \beta \to ((\text{APVAR} \to \text{TERM}) \cup (\text{SLVAR} \to \beta))$
$\to \text{TERM} \to \text{TERM}$ }

subst $\to$ Combinator {**type:** Assign $\to$ TERM $\to$ TERM}

Hiddens

Context-free Functions

def-list $\to$ Combinator

Variables

*[FL] [0-9']** $\to$ Term

Equations

Given some assignment ρ we replace all occurrences of variables in a term t by their value in ρ trough the substitution function *substp*.

$$\begin{aligned} \mathit{substp}\ F\ L\ \rho\ x^{(*)} &= \rho\ x^{(*)} \\ \mathit{substp}\ F\ L\ \rho\ f(\vec{t}) &= \mathit{app}\ f\ (\mathit{map}\ (\mathit{substp}\ F\ L\ \rho)\ [\![\vec{t}]\!]_{\vec{t}}) \\ \mathit{substp}\ F\ L\ \rho\ [\vec{t}] &= \mathit{foldr}\ F\ L\ (\mathit{zip}\ \langle [\![\vec{t}]\!]_{\vec{t}},\ \mathit{map}\ (\mathit{substp}\ F\ L\ \rho)\ [\![\vec{t}]\!]_{\vec{t}} \rangle) \end{aligned}$$

The default substitution function *subst* is implemented using *def-list* and the empty list as parameters to *substp*.

$$\mathit{subst}\ \rho\ t = \mathit{substp}\ \mathit{def\text{-}list}\ s\ \rho\ t \Leftarrow s = []$$

where *def-list* is defined as

$$\textit{def-list}\ \langle x^*, [\![\vec{t}]\!]_{\vec{t}}\rangle\ [\vec{s}] = [\vec{t}, \vec{s}] \qquad \textit{def-list}\ \langle x, t\rangle\ [\vec{s}] = [t, \vec{s}]$$

Subst expects that all sublist variables are mapped to lists of TERMs.

11.3 Matching

Imports Assignments$^{(11.1)}$ RN2-Terminology$^{(9.4)}$

Now, we define the function *match* that, given an open term t, a closed term s and an assignment ρ, yields the list of all assignments ρ' such that ρ' is an extension of ρ and $(\textit{subst}\ \rho'\ t \equiv s)$.

If the list of assignments is empty, matching failed. More than one assignment can result if t_1 contains an associative list expression.

Exports
  Context-free Functions
    match $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow \mathbb{B} \times$ Assign*}
Hiddens
  Context-free Functions
    match-t $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow$ Assign*}
    add $\rightarrow$ Combinator {**type:** Assign $\rightarrow$ Assign* $\rightarrow$ Assign*}
  Variables
    [*X*] [*0-9'*]* $\rightarrow$ Term

Equations

Match Given an open term t, a closed term s and an assigment ρ *match* computes a pair $\langle \mathcal{B}, [\vec{\rho}]\rangle$ such that $[\vec{\rho}]$ is the list of all extensions of ρ ρ' for which

$$\textit{subst}\ \rho'\ t \equiv s\ .$$

If no such asignments exist $\mathcal{B}$ is $\perp$ otherwise $\top$.

$$\frac{[\vec{\rho}] = \textit{match-t}\ t\ s\ \rho}{\textit{match}\ t\ s\ \rho = \langle \neg\ (\varepsilon ?\ [\vec{\rho}]),\ [\vec{\rho}]\rangle}$$

Sets of Assignments In building a list of all matching assignments we are not interested in failing assignments that result from adding a pair which make the assignment inconsistent.

$$\textit{add}\ \perp_\rho\ [\vec{\rho}] = [\vec{\rho}] \qquad \textit{add}\ \rho\ [\vec{\rho}] = [\rho, \vec{\rho}] \Leftarrow \rho \neq \perp_\rho$$

Match-t *Match* is implemented by *match-t* which, given a closed term, an open term and an assignment, determines all matches extending that assignment.

Fail Cases An error assignment $(\perp_\rho)$ can not be extended to a matching assignment.

$$\textit{match-t}\ X_1\ X_2\ \perp_\rho = []$$

Applications with different function symbols do not match.

$$\textit{match-t}\ f(\vec{t})\ g(\vec{s})\ \rho = [] \Leftarrow f \neq g$$

Lists and applications do not match.

$$\textit{match-t}\ [\vec{t}]\ f(\vec{s})\ \rho = [] \qquad \textit{match-t}\ f(\vec{t})\ [\vec{s}]\ \rho = []$$

Variables A normal variable matches with any term. The asignment $(x \mapsto t)$ is added to ρ.

$$match\text{-}t\ x\ t\ \rho = add\ (x \mapsto t \oplus \rho)\ [] \Leftarrow \rho \neq \bot_\rho$$

Note that if a term was already assigned to x in ρ, this composition of assignments yields $\bot_\rho$. This means that non-linear patterns can be used for matching with the proper meaning.

Applications and Lists If the terms are both applications with the same function symbol, or both lists, we have to match the subterms of the terms.

$$\begin{aligned} match\text{-}t\ f(\vec{t})\ f(\vec{s})\ \rho &= match\text{-}t\ \llbracket \vec{t} \rrbracket_{\vec{t}}\ \llbracket \vec{s} \rrbracket_{\vec{t}}\ \rho \Leftarrow \rho \neq \bot_\rho \\ match\text{-}t\ [\vec{t}]\ [\vec{s}]\ \rho &= match\text{-}t\ \llbracket \vec{t} \rrbracket_{\vec{t}}\ \llbracket \vec{s} \rrbracket_{\vec{t}}\ \rho \Leftarrow \rho \neq \bot_\rho \end{aligned}$$

Lists of Terms The arguments of an application or list are matched by the same method. The only distinction is that list patterns may contain sublist variables, which may not be the arguments of applications.

When the pattern and term are the empty list, matching succeeds.

$$match\text{-}t\ \llbracket\rrbracket_{\vec{t}}\ \llbracket\rrbracket_{\vec{t}}\ \rho = [\rho] \Leftarrow \rho \neq \bot_\rho$$

If either the pattern or the term has superfluous terms matching fails. This can happen only if pattern and term were lists. Two applications of the same function always have the same number of arguments by the static constraints on TRSs.

$$match\text{-}t\ \llbracket\rrbracket_{\vec{t}}\ \llbracket s,\ \vec{s} \rrbracket_{\vec{t}}\ \rho = [] \qquad match\text{-}t\ \llbracket t,\ \vec{t} \rrbracket_{\vec{t}}\ \llbracket\rrbracket_{\vec{t}}\ \rho = [] \Leftarrow type\ t \neq slTp$$

A list pattern with a non sublist-variable as first element.

$$\frac{\rho \neq \bot_\rho,\ \ type\ t \neq slTp}{match\text{-}t\ \llbracket t,\ \vec{t} \rrbracket_{\vec{t}}\ \llbracket s,\ \vec{s} \rrbracket_{\vec{t}}\ \rho = flatMap\ (match\text{-}t\ \llbracket \vec{t} \rrbracket_{\vec{t}}\ \llbracket \vec{s} \rrbracket_{\vec{t}})\ (match\text{-}t\ t\ s\ \rho)}$$

Sublist Variables (List Matching) When the first variable of the pattern is a sublist variable x^*, several matches are possible since x^* can match with any prefix of $\llbracket \vec{s} \rrbracket_{\vec{t}}$. We use a list-comprehension to enumerate all prefixes. The variable $\_i$ ranges over the interval $0 \ldots n$, where n is the length of $\llbracket \vec{s} \rrbracket_{\vec{t}}$. Then x^* is assigned to the first $\_i$ terms of $\llbracket \vec{s} \rrbracket_{\vec{t}}$, and the rest of the pattern is matched against the rest of the terms resulting in a list of assignments. The whole comprehension yields a list of lists of assignments. These are flattened into a list of assignments by *concat*.

$$\frac{\rho \neq \bot_\rho}{\begin{array}{l} match\text{-}t\ \llbracket x^*,\ \vec{t} \rrbracket_{\vec{t}}\ \llbracket \vec{s} \rrbracket_{\vec{t}}\ \rho\ = \\ concat\ [match\text{-}t\ \llbracket \vec{t} \rrbracket_{\vec{t}}\ (drop\ \_i\ \llbracket \vec{s} \rrbracket_{\vec{t}})\ (x^* \mapsto take\ \_i\ \llbracket \vec{s} \rrbracket_{\vec{t}} \oplus \rho) \mid \_i \leftarrow 0\ ..\ \#\ \llbracket \vec{s} \rrbracket_{\vec{t}}] \end{array}}$$

Example The table below shows the assignments resulting from

$$match\ t\ [a(), b(), a(), b()]\ (\Delta_\rho\ \mathbf{I})$$

for various list patterns t.

Pattern t	Assigments			
$[X, X*]$	$(X* \mapsto \llbracket b(), a(), b() \rrbracket_{\vec{t}}) \oplus (X \mapsto a())$			$\oplus\ (\Delta_\rho\ \mathbf{I})$
$[X*, X]$	$(X* \mapsto \llbracket a(), b(), a() \rrbracket_{\vec{t}}) \oplus (X \mapsto b())$			$\oplus\ (\Delta_\rho\ \mathbf{I})$
$[X*, X, Y*]$	$(X* \mapsto \llbracket\rrbracket_{\vec{t}})$	$\oplus\ (X \mapsto a())$	$\oplus\ (Y* \mapsto \llbracket b(), a(), b() \rrbracket_{\vec{t}})$	$\oplus\ (\Delta_\rho\ \mathbf{I})$
	$(X* \mapsto \llbracket a() \rrbracket_{\vec{t}})$	$\oplus\ (X \mapsto b())$	$\oplus\ (Y* \mapsto \llbracket a(), b() \rrbracket_{\vec{t}})$	$\oplus\ (\Delta_\rho\ \mathbf{I})$
	$(X* \mapsto \llbracket a(), b() \rrbracket_{\vec{t}})$	$\oplus\ (X \mapsto a())$	$\oplus\ (Y* \mapsto \llbracket b() \rrbracket_{\vec{t}})$	$\oplus\ (\Delta_\rho\ \mathbf{I})$
	$(X* \mapsto \llbracket a(), b(), a() \rrbracket_{\vec{t}})$	$\oplus\ (X \mapsto b())$	$\oplus\ (Y* \mapsto \llbracket\rrbracket_{\vec{t}})$	$\oplus\ (\Delta_\rho\ \mathbf{I})$

Claim 11.3.1 *For closed terms s and t*

$$\pi\ 1\ (\mathit{match}\ s\ t\ (\Delta_\rho\ \mathbf{I})) \leftrightarrow s \equiv t$$

That is closed terms only match when they are syntactically equal.

11.4 Rule Segments

Imports RN2-Terminology[(9.4)]

A TRS is translated to a table that maps a function symbol to the list (or *segment*) of rules that define that function symbol. We will use the following type abbreviations for segments and rule tables

$$\mathbb{S}_\varphi = \text{FUNSYM} \times \text{RULE}^* \qquad \mathbb{R} = \mathbb{S}_\varphi^*$$

Exports

Context-free Functions

"$\mathit{segment}_{\vec{\varphi}}$" $\rightarrow$ Combinator {**type:** TRS $\rightarrow \mathbb{R}$}

Variables

"$\mathcal{S}$" *[0-9′]*∗ $\rightarrow$ Term

"$\vec{\mathcal{S}}$" *[0-9′]*∗ $\rightarrow$ {Term "," }∗

"$\mathcal{R}$" $\rightarrow$ Term

We will use the variable $\mathcal{R}$ for a table of segments, $\mathcal{S}$ for segments and $\vec{\mathcal{S}}$ for lists of segments.

Equations

$\mathit{segment}_{\vec{\varphi}}$ divides the rules of a specification into a list of segments. A segment is a pair $(f, [\vec{\varphi}])$ of a function symbol and the rules that define that function symbol.

$$\begin{aligned} \mathit{segment}_{\vec{\varphi}}\ \mathcal{T} \ =\ & [\langle \_f,\ \mathit{sort}\ (\leq)\ (\downarrow_{\vec{\varphi}}\ [\_\mathit{rule} \mid \_\mathit{rule} \leftarrow \mathbf{R}\ \mathcal{T};\ \mathit{if}\ \mathit{fun}\ \_\mathit{rule} \equiv \_f])\rangle \\ & \mid \_f \leftarrow \mathit{map}\ \mathit{fun}\ (\downarrow_{[]}\ (\Sigma\ \mathcal{T}))] \end{aligned}$$

The outer comprehension ranges over all function symbols declared in the signature of $\mathcal{T}$ and constructs a segment for each function symbol. The inner comprehension selects the rules defining $\_f$.

Note that the rules are sorted in descending order of priority (rules with a lower *ord* have higher priority; see module 9.4.6). This means that default rules come after normal rules, but otherwise the order of the rules in the specification is preserved. For instance, suppose φ_i, for $1 \leq i \leq 4$, are normal rules that define f, and that a specification contains the rules in the order

$$\varphi_1\ \ \mathit{otherwise}\colon \varphi_2\ \ \varphi_3\ \ \mathit{otherwise}\colon \varphi_4$$

then the segment for f will be

$$\langle f, \varphi_1\ \ \varphi_3\ \ \mathit{otherwise}\colon \varphi_2\ \ \mathit{otherwise}\colon \varphi_4\rangle$$

11.5 Term Reduction

Imports Matching[(11.3)] Rule-Segments[(11.4)]

We define the normalization of a closed term in the context of a TRS. Reduction of a term in a TRS is expressed by the function

Exports

Context-free Functions

TRS "⟦" TERM "⟧" $\rightarrow$ TERM

The reduction strategy is leftmost-innermost. The algorithm is essentially the same as that described in [Wal91]. Apart from the differences mentioned earlier, this specification can not use global variables. Therefore the rule-table and the assignments are arguments of the functions.

Hiddens

Context-free Functions

normalize	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ TERM $\rightarrow$ TERM}
reduce1	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ TERM $\rightarrow$ TERM}
reduce	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ RULE* $\rightarrow$ TERM $\rightarrow$ TERM}
inst	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ Assign $\rightarrow$ TERM $\rightarrow$ TERM}
val-cond	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ COND $\rightarrow$ Assign $\rightarrow$ Assign*}
validate	$\rightarrow$ Combinator	{**type:** $\mathbb{R} \rightarrow$ COND* $\rightarrow$ Assign* $\rightarrow$ Assign*}

Equations

The normal form of a term t in a TRS $\mathcal{T}$ is the term that results from *normalizing* it with the rule-table of $\mathcal{T}$.

$$\mathcal{T}[\![t]\!] = t' \Leftarrow t' = \mathit{normalize}\ (\mathit{segment}_{\vec{\varphi}}\ \mathcal{T})\ t$$

where normalization of a term is a postorder traversal of the term that reduces all subterms.

$$\mathit{normalize}\ \mathcal{R}\ t = \mathit{postorder}\ (\mathit{reduce1}\ \mathcal{R})\ t$$

This means that $(\mathit{reduce1}\ \mathcal{R})$ is applied to each subterm s of t (including t) after the proper subterms of s have been normalized.

Reduce *Reduce1* gets a rule-table and a closed term t, all of whose proper subterms are in normal form, and computes its normal form. A closed term is either a list or an application. Since lists may not be the lhs of a rule, a list, with all its subterms in nf, is itself in nf.

$$\mathit{reduce1}\ \mathcal{R}\ [\vec{t}] = [\vec{t}]$$

An application of a function f is reduced by looking up the segment for f in the rule-table and calling *reduce* that checks all rules in order until a rule φ is found such that the lhs of φ matches the application and the conditions can be validated.

$$\mathit{reduce1}\ \mathcal{R}\ f(\vec{t}) = \mathit{reduce}\ \mathcal{R}\ (\mathcal{R}\ .\ f)\ f(\vec{t})$$

If the segment for f is empty there are no (more) rules that define f; $f(\vec{t})$ is in normal form.

$$\mathit{reduce}\ \mathcal{R}\ [\![]\!]_{\vec{\varphi}}\ f(\vec{t}) = f(\vec{t})$$

If there is at least one rule φ for f, we try to match $f(\vec{t})$ with the left-hand side of φ and if that succeeds we try to validate the conditions of φ. If at least one assignment results from the validation, the rule applies to $f(\vec{t})$ and we instantiate the rhs of φ with the first assignment ρ.

$$\frac{\begin{array}{ll}\langle\top, [\vec{\rho}]\rangle = \mathit{match}\ (\mathit{lhs}\ \varphi)\ f(\vec{t})\ (\Delta_\rho\ \mathbf{I}), \\ [\rho, \vec{\rho}\,'] \ = \mathit{validate}\ \mathcal{R}\ (\mathit{conds}\ \varphi)\ [\vec{\rho}]\end{array}}{\mathit{reduce}\ \mathcal{R}\ [\![\varphi\ \vec{\varphi}]\!]_{\vec{\varphi}}\ f(\vec{t}) = \mathit{inst}\ \mathcal{R}\ \rho\ (\mathit{rhs}\ \varphi)}$$

If the previous equation did not succeed, either by failing to match the lhs or by failing to validate one of the conditions, the rest of the rules for f are tried.

$$\mathit{reduce}\ \mathcal{R}\ [\![\varphi\ \vec{\varphi}]\!]_{\vec{\varphi}}\ f(\vec{t}) = \mathit{reduce}\ \mathcal{R}\ [\![\vec{\varphi}]\!]_{\vec{\varphi}}\ f(\vec{t}) \quad \textbf{otherwise}$$

Instantiate Given an assignment ρ and an open term t, *inst* substitutes the the variables in t by their values in ρ and normalizes the resulting term. That is

$$\mathit{inst}\ \mathcal{R}\ \rho\ t = \mathit{normalize}\ \mathcal{R}\ (\mathit{subst}\ \rho\ t) \qquad \bigstar$$

But, making use of the fact that the terms substituted for the variables in t are already in nf, a more efficient specification of *inst* is

$$\begin{aligned}
&\mathit{inst}\ \mathcal{R}\ \rho\ x &&= \rho\ x \\
&\mathit{inst}\ \mathcal{R}\ \rho\ f(\vec{t}) &&= \mathit{reduce1}\ \mathcal{R}\ (\mathit{app}\ f\ (\mathit{map}\ (\mathit{inst}\ \mathcal{R}\ \rho)\ [\![\vec{t}]\!]_{\vec{t}})) \\
&\mathit{inst}\ \mathcal{R}\ \rho\ [\vec{t}] &&= \mathit{list}\ (\mathit{inst}\ \mathcal{R}\ \rho\ [\![\vec{t}]\!]_{\vec{t}})
\end{aligned}$$

$$\begin{aligned}
&\mathit{inst}\ \mathcal{R}\ \rho\ [\![]\!]_{\vec{t}} &&= [\![]\!]_{\vec{t}} \\
&\mathit{inst}\ \mathcal{R}\ \rho\ [\![x^*, \vec{t}]\!]_{\vec{t}} &&= \rho\ x^*\ +\!\!+\ \mathit{inst}\ \mathcal{R}\ \rho\ [\![\vec{t}]\!]_{\vec{t}} \\
&\mathit{inst}\ \mathcal{R}\ \rho\ [\![t, \vec{t}]\!]_{\vec{t}} &&= \mathit{inst}\ \mathcal{R}\ \rho\ t : \mathit{inst}\ \mathcal{R}\ \rho\ [\![\vec{t}]\!]_{\vec{t}} \Leftarrow \mathit{type}\ t \neq \mathit{slTp}
\end{aligned}$$

Validate Condition We now have defined all ingredients of term reduction, except for the validation of conditions. We start with the validation of a single condition. Positive and negative conditions are handled similarly. The lhs and rhs are instantiated (and thus reduced to normal form) and the resulting normal forms are compared by $(\equiv)$. A positive conditions states that the nf's of lhs and rhs should be equal, a negative condition states that they should be unequal. Failure is indicated by an empty list of assignments.

$$\begin{aligned}
&\mathit{val\text{-}cond}\ \mathcal{R}\ s \stackrel{?}{=} t\ \rho = \textbf{\textit{if}}\ (\mathit{inst}\ \mathcal{R}\ \rho\ s \equiv \mathit{inst}\ \mathcal{R}\ \rho\ t)\ [\rho]\ [] \\
&\mathit{val\text{-}cond}\ \mathcal{R}\ s \neq t\ \rho = \textbf{\textit{if}}\ (\neg\ (\mathit{inst}\ \mathcal{R}\ \rho\ s \equiv \mathit{inst}\ \mathcal{R}\ \rho\ t))\ [\rho]\ []
\end{aligned}$$

A condition that introduces variables is handled differently. The rhs is instantiated and matched, extending the current assignment ρ, against the lhs. If matching succeeded, the list of assignments, resulting from the match, is returned.

$$\frac{\langle\mathcal{B}, [\vec{\rho}]\rangle = \mathit{match}\ s\ (\mathit{inst}\ \mathcal{R}\ \rho\ t)\ \rho}{\mathit{val\text{-}cond}\ \mathcal{R}\ s := t\ \rho = \textbf{\textit{if}}\ \mathcal{B}\ [\vec{\rho}]\ []}$$

Validate Conditions The last step is the validation of a list of conditions. Validation of the conditions of a rule starts with the list of conditions and the list of assignments resulting from the match with the lhs of the rule. If there are no (more) conditions to validate, validation succeeds.

$$\mathit{validate}\ \mathcal{R}\ [\![]\!]_{\vec{C}}\ [\vec{\rho}] = [\vec{\rho}]$$

If no assignments remain validation fails.

$$\mathit{validate}\ \mathcal{R}\ [\![C, \vec{C}]\!]_{\vec{C}}\ [\,] = [\,]$$

If there is at least one assignment ρ and one condition C, then try validating C with ρ and the rest of the conditions with the resulting list of assignments. If this fails try to validate C and the rest of the conditions with the rest of the assignments. The assignments resulting from these two trials are concatenated into one list of assignments.

$$\begin{array}{rcl} \mathit{validate}\ \mathcal{R}\ [\![C, \vec{C}]\!]_{\vec{C}}\ [\rho, \vec{\rho}] & = & \mathit{validate}\ \mathcal{R}\ [\![\vec{C}]\!]_{\vec{C}}\ (\mathit{val\text{-}cond}\ \mathcal{R}\ C\ \rho) \\ & & + \! + \\ & & \mathit{validate}\ \mathcal{R}\ [\![C, \vec{C}]\!]_{\vec{C}}\ [\vec{\rho}] \end{array}$$

Remark In this definition we saw that the lhs of a rule, and the lhs of a newvars condition are used as patterns for matching, whereas the rhs of a rule, the rhs of a newvars condition, and both sides of a positive or negative condition are instantiated. In the future we will refer to these categories of terms as, respectively, *matching terms* and *instantiating terms.*

Example Let

$$\begin{array}{rl} \mathcal{T} = & \mathit{signature}\ a()\ b() \\ & \mathit{equations} \\ & \quad [a]\ a() \stackrel{?}{=} b() ===> a() = b() \end{array}$$

What is the nf of $a()$, i.e. what is $\mathcal{T}[\![a()]\!]$?

Since $a()$ matches with the lhs of rule $[a]$, $a()$ would reduce to $b()$ if the condition $a() \stackrel{?}{=} b()$ is validated. To decide this the nf of $a()$ has to be found.

Conclusion: $\mathcal{T}[\![a()]\!]$ builds an infinite stack of conditions to be validated and never terminates.

Chapter 12

Transformation LM

In the previous chapter we saw that associative lists lead to a complex matching mechanism. List patterns with more than one sublist variable may have more than one match. In ARM the assignments for a list pattern are not computed all at once as we did in the definition of *match*. Instead, a suitable assignment is computed and validation of (the rest of) the conditions is done. If that fails the next assignment is computed. This implies a considerable amount of bookkeeping in the machine.

In this and the next two chapters we will show that a TRS with associative lists can be transformed to a TRS with lists represented by the constructors *$cons* and *$nil*. This entails that associative lists do not enhance the computational power of RN2.

The TRSs that result from this transformation can be compiled into ARM programs that do not need the primitives for list matching. Another advantage of the elimination of associative lists is that garbage collection can be simplified.

Plan A specification is transformed in three phases. The first phase, transformation LL, removes non-left-linearity from a specification. The second phase, transformation FL, flattens list expressions in rewrite rules, such that list patterns with more than one sublist variable only occur as top terms of lhs's of assignment conditions. The third phase, transformation LM, eliminates associative lists, replacing them by *$cons*/*$nil* lists.

First we will specify the most complicated of the transformations, transformation LM. Therefore, we assume that the TRSs to be transformed are left-linear and flat. In Chapters 13 and 14 we will then deal with flattening lists and making TRSs left-linear.

We start in Section 12.1 with the definition of the RN2 signature of *$cons*/*$nil* lists extended with auxiliary RN2 functions that will be needed in the transformed TRSs. In Section 12.2 we look at some example transformations. The rest of the chapter consists of the definition of the transformation.

The import graph for this chapter is shown in figure 12.1.

12.1 Representing Lists by Cons and Nil

Imports RN2-Terminology$^{(9.4)}$ Substitution$^{(11.2)}$

The TRS $\mathcal{T}_{cons}$ will be used to extend transformed TRSs with *$cons*/*$nil* lists. The TRS includes the constructors *$cons* and *$nil* and several auxiliary functions used in the rules generated by transformation *lm*.

Exports

Context-free Functions

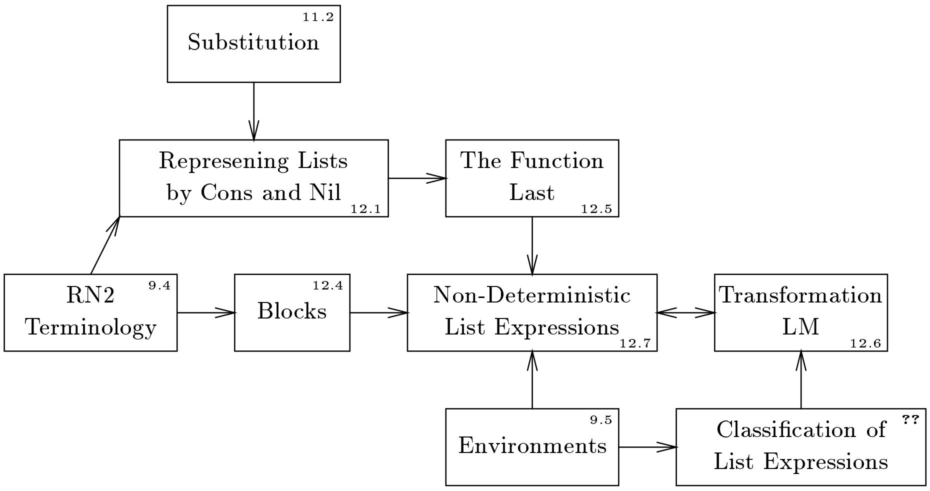


Figure 12.1: Import graph for Chapter 12; Transformation LM

"$\mathcal{T}_{cons}$" $\rightarrow$ TRS

Then we define the following combinators for the construction of terms over the signature of $\mathcal{T}_{cons}$.

Exports

Context-free Functions

"*Nil*" $\rightarrow$ Combinator {**type:** TERM}
"*Cons*" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow$ TERM}
"*Fail*" $\rightarrow$ Combinator {**type:** TERM}
"*Tl*" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM}
"*Nth-Tl*" $\rightarrow$ Combinator {**type:** $\mathbb{N} \rightarrow$ TERM $\rightarrow$ TERM}
"*Conc*" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow$ TERM}
"*Until*" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow$ TERM}
"*ConcUntil*" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM $\rightarrow$ TERM $\rightarrow$ TERM}

The combinator $cons_t$ translates the associative lists in a term to *\$cons*/*\$nil* lists. The combinator $cons_t^{-1}$ is the inverse of $cons_t$ for ground terms; it translates terms with *\$cons*/*\$nil* lists to terms with associative lists.

Exports

Context-free Functions

"$cons_t$" $\rightarrow$ Combinator {**type:** (SLVAR $\rightarrow$ TERM) $\rightarrow$ TERM $\rightarrow$ TERM}
"$cons_t^{-1}$" $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM}
"*List*" $\rightarrow$ Combinator {**type:** (SLVAR $\rightarrow$ TERM) $\rightarrow$ TERM$*$ $\rightarrow$ TERM}

In the definition of $cons_t$ the following auxiliary functions are used.

Hiddens

Context-free Functions

ct $\rightarrow$ Combinator {**type:** (SLVAR $\rightarrow$ TERM) $\rightarrow$ TERM $\rightarrow$ TERM}
fold-cn $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM}
fold-cn-c $\rightarrow$ Combinator {**type:** TERM $\rightarrow$ TERM}

Variables

[*F*] [*0-9'*]$*$ $\rightarrow$ Term

Equations

The TRS $\mathcal{T}_{cons}$ declares the constructors *\$cons* and *\$nil* that will be used to represent associative lists in transformed TRSs. In addition to these constructors, auxiliary functions are defined that are used in the rules generated for non-deterministic patterns.

$$
\begin{array}{ll}
\mathcal{T}_{cons} \; = & \\
\quad \textit{signature} & \textit{\$nil}() \\
& \textit{\$cons}(\textit{appTp},\ \textit{appTp}) \\
& \textit{\$fail}() \\
& \textit{\$tl}(\textit{appTp}) \\
& \textit{\$conc}(\textit{appTp},\ \textit{appTp}) \\
& \textit{\$until}(\textit{appTp},\ \textit{appTp}) \\
& \textit{\$concUntil}(\textit{appTp},\ \textit{appTp},\ \textit{appTp}) \\
& \textit{\$eq}(\textit{appTp},\ \textit{appTp}) \\
& \textit{\$yes}() \\
& \textit{\$no}() \\
\quad \textit{equations} & [\textit{\$tl}]\ \textit{\$tl}(\textit{\$cons}(X,\ L)) = L \\
& [\textit{\$conc0}]\ \textit{\$conc}(\textit{\$nil}(),\ L) = L \\
& [\textit{\$conc1}]\ \textit{\$conc}(L,\ \textit{\$nil}()) = L \\
& [\textit{\$conc2}]\ \textit{\$conc}(\textit{\$cons}(X,\ L1),\ L2) = \textit{\$cons}(X,\ \textit{\$conc}(L1,\ L2)) \\
& [\textit{\$cu}]\ \textit{\$conc}(\textit{\$until}(L1,\ L2),\ L3) = \textit{\$concUntil}(L1,\ L2,\ L3) \\
& [\textit{\$unt0a}]\ \textit{\$until}(\textit{\$nil}(),\ L2) = L2 \\
& [\textit{\$unt0b}]\ \textit{\$eq}(L1,\ L2) \stackrel{?}{=} \textit{yes}() ===> \textit{\$until}(L1,\ L2) = \textit{\$nil}() \\
& [\textit{\$untn}]\ \textit{\$eq}(L1,\ L2) \stackrel{?}{=} \textit{no}(),\ \textit{\$cons}(I,\ L3) := L2 \\
& \qquad\quad ===> \\
& \qquad\quad \textit{\$until}(L1,\ L2) = \textit{\$cons}(I,\ \textit{\$until}(L1,\ L3)) \\
& [\textit{\$concUnt0}]\ \textit{\$concUntil}(\textit{\$nil}(),\ L2,\ L3) = \textit{\$conc}(L2,\ L3) \\
& [\textit{\$concUnt0}]\ \textit{\$eq}(L1,\ L2) \stackrel{?}{=} \textit{yes}() \\
& \qquad\qquad ===> \textit{\$concUntil}(L1,\ L2,\ L3) = L3 \\
& [\textit{\$concUntn}]\ \textit{\$eq}(L1,\ L2) \stackrel{?}{=} \textit{no}(),\ \textit{\$cons}(I,\ L4) := L2 \\
& \qquad\qquad ===> \\
& \qquad\qquad \textit{\$concUntil}(L1,\ L2,\ L3) = \textit{\$cons}(I,\ \textit{\$concUntil}(L1,\ L4,\ L3)) \\
& [\textit{\$eqY}]\ X \stackrel{?}{=} Y ===> \textit{\$eq}(X,\ Y) = \textit{yes}() \\
& \textit{otherwise} : [\textit{\$eqN}]\ \textit{\$eq}(X,\ Y) = \textit{no}()
\end{array}
$$

Most of the functions in $\mathcal{T}_{cons}$ are standard list functions. Lists are constructed with *\$cons* and *\$nil*. The tail of a non-empty list is computed by *\$tl*. Lists are concatenated with *\$conc*. The constant *\$fail*() will be used to indicate failure.

The last two functions are non-standard and deserve special attention. The function *\$until* has two list arguments $L1$ and $L2$. The first argument should be a tail of the second. That is, for some $n \geq 0$, $L1 = \textit{\$tl}(\ldots \textit{\$tl}(L2)\ldots)$, where the dots denote n applications of *\$tl*. The result of $\textit{\$until}(L1, L2)$ then is the prefix of $L2$ until the list $L1$ is reached. The function *\$concUntil* does the same as *\$until* but concatenates the prefix directly to its third list argument. This prevents the construction of a list with *\$until* that is subsequently concatenated to another list, which entails that the list is constructed again, causing unnecessary garbage.

The equality predicate *\$eq*, used in the definition of *\$until* and *\$concUntil*, is expensive if both lists are equal or have a large prefix in common. The lists have to be traversed and all elements compared. However, the functions *\$until* and *\$concUntil* will be used in special circumstances. The first argument of *\$until* will always be a subterm of the second argument. This means that, in the representation in the ARM machine, the *\$eq* predicate can be implemented by comparing the pointers to $L1$ and $L2$.

Constructing cons/nil Lists We define combinators to construct terms over the signature of $\mathcal{T}_{cons}$.

$$\begin{array}{ll} \mathit{Cons}\ t_1\ t_2 = \mathit{\$cons}(t_1, t_2) & \mathit{Nil}\ = t \Leftarrow t = \mathit{\$nil}() \\ \mathit{Tl}\ t \qquad = \mathit{\$tl}(t) & \mathit{Fail} = t \Leftarrow t = \mathit{\$fail}() \\ \mathit{Conc}\ t_1\ t_2 = \mathit{\$conc}(t_1, t_2) & \\ \mathit{Until}\ t_1\ t_2 = \mathit{\$until}(t_1, t_2) & \end{array}$$

$$\mathit{ConcUntil}\ t_1\ t_2\ t_3 = \mathit{\$concUntil}(t_1, t_2, t_3)$$

The function $(\mathit{Nth\text{-}Tl}\ n)$ constructs, for a term t, the term $\mathit{\$tl}(\ldots \mathit{\$tl}(t)\ldots)$, that is, $\mathit{\$tl}$ applied n times to t.

$$\mathit{Nth\text{-}Tl}\ n = \mathit{Tl}\ \hat{}\ n$$

From Associative List to cons/nil List TERMs with associative lists are translated to cons/nil lists by $cons_t$, which is parameterized with an assignment. The term constructed is optimized by *fold-cn* (see below).

$$cons_t\ F\ t = \mathit{fold\text{-}cn}\ (\mathit{substp}\ ct\ \mathit{Nil}\ F\ t)$$

where ct is defined as

$$\begin{array}{l} ct\ \langle a, b\rangle\ c\ \ = \mathit{\$cons}(b, c) \Leftarrow \mathit{type}\ a \neq \mathit{slTp} \\ ct\ \langle x^*, b\rangle\ c = \mathit{\$conc}(b, c) \end{array}$$

We need this function to translate the terms in rewrite rules of TRSs and to preprocess ground terms that are to be rewritten using a transformed TRS.

The combinator *List* is a shortcut for $cons_t$ for lists of terms.

$$\mathit{List}\ F\ [\![\vec{t}\,]\!]_{\vec{t}} = cons_t\ F\ [\vec{t}\,]$$

Example The ground term $[a(), b(), c()]$ is translated to the obvious representation with *\$cons* and *\$nil*.

$$cons_t\ \mathbf{I}\ [a(), b(), c()] \twoheadrightarrow \mathit{\$cons}(a(), \mathit{\$cons}(b(), \mathit{\$cons}(c(), \mathit{\$nil})))$$

The list $[a(), X*, b()]$ is translated as follows

$$cons_t\ (\backslash *)\ [a(), X*, b()] \twoheadrightarrow \mathit{\$cons}(a(), \mathit{\$conc}(X\$, \mathit{\$cons}(b(), \mathit{\$nil}))$$

Note that the $*$ is replaced by a \$ by $(\backslash *)$.

If we assign to the variable $X*$ in the previous example the term $\mathit{\$until}(X, Y)$ we see the following

$$\begin{array}{l} cons_t\ (X* \mapsto \mathit{\$until}(X, Y))\ [a(), X*, b()] \\ \twoheadrightarrow \mathit{fold\text{-}cn}\ \mathit{\$cons}(a(), \mathit{\$conc}(\mathit{\$until}(X, Y), \mathit{\$cons}(b(), \mathit{\$nil})) \\ \twoheadrightarrow \mathit{\$cons}(a(), \mathit{\$concUntil}(X, Y, \mathit{\$cons}(b(), \mathit{\$nil})) \end{array}$$

First the list is straightforwardly translated to a *\$cons*/*\$nil* list and then the combination of *\$conc* and *\$until* is translated to *\$concUntil* by *fold-cn*.

And Back The inverse of $cons_t$, $cons_t^{-1}$ maps closed BRN2 TERMs with cons/nil lists to RN2 TERMs with associative lists. $cons_t^{-1}$ maps *\$nil* to []

$$cons_t^{-1}\ t = s \Leftarrow t = \$nil(),\ \ s = []$$

and maps $\$cons(X, L)$ to $[X', X*]$, if X is mapped to X' and L is mapped to $[X*]$.

$$cons_t^{-1}\ \$cons(a,\, b) = list\ (cons_t^{-1}\ a : items\ (cons_t^{-1}\ b))$$

If the term is not an application of *\$nil* or *\$cons* the subterms are converted and the application is restored.

$$cons_t^{-1}\ f(\vec{t}) = app\ f\ (map\ cons_t^{-1}\ [\![\vec{t}]\!]_{\vec{t}}) \Leftarrow f \neq \$cons,\ \ f \neq \$nil$$

Example

$$cons_t^{-1}\ \$cons(a(), \$cons(b(), \$nil)) \twoheadrightarrow [a(), b()]$$

$$\begin{aligned}&cons_t^{-1}\ sum(\$cons(zero(), \$cons(succ(zero()), \$nil)))\\ &\twoheadrightarrow sum([zero(), succ(zero())])\end{aligned}$$

For closed terms t with associative lists we have

$$(cons_t^{-1} \circ cons_t\ \mathbf{I})\ t = t \qquad \bigstar$$

Partial Evaluation of cons/nil Lists Using the rules of specification $\mathcal{T}_{cons}$, *fold-cn* (fold cons/nil) reduces terms (possibly containing variables) over $(\Sigma\ \mathcal{T}_{cons})$ as much as possible. This is useful since it makes the definition of $cons_t$ straightforward. For instance, $(cons_t\ (\backslash *))$ translates $[X*]$ first to $\$conc(X\$, \$nil)$. By the rules in $\mathcal{T}_{cons}$ this is equivalent to $X\$$. This last transformation is carried out by *fold-cn*. At each node in the tree t it applies *fold-cn-c*.

$$fold\text{-}cn\ t = postorder\ fold\text{-}cn\text{-}c\ t$$

The cases are dealt with by *fold-cn-c*.

$$\begin{array}{llll} fold\text{-}cn\text{-}c\ x^{(*)} & = x^{(*)} & fold\text{-}cn\text{-}c\ \$tl(\$cons(a,\, b)) & = b\\ fold\text{-}cn\text{-}c\ \$conc(\$nil(),\, a) & = a & fold\text{-}cn\text{-}c\ \$conc(a,\, \$nil()) & = a\\ fold\text{-}cn\text{-}c\ \$until(\$nil(),\, a) & = a & fold\text{-}cn\text{-}c\ \$until(a,\, a) & = Nil \end{array}$$

$$\begin{aligned} fold\text{-}cn\text{-}c\ \$conc(\$cons(a,\, b),\, c) &= Cons\ a\ (fold\text{-}cn\text{-}c\ \$conc(b,\, c))\\ fold\text{-}cn\text{-}c\ \$conc(\$until(a,\, b),\, c) &= fold\text{-}cn\text{-}c\ \$concUntil(a,\, b,\, c)\\ fold\text{-}cn\text{-}c\ \$concUntil(a,\, a,\, c) &= c\\ fold\text{-}cn\text{-}c\ \$concUntil(a,\, b,\, \$nil()) &= \$until(a,\, b) \end{aligned}$$

When none of these rules apply, there is nothing to fold.

$$fold\text{-}cn\text{-}c\ f(\vec{t}) = f(\vec{t}) \quad \textbf{otherwise}$$

Remark *Fold-cn* applies innermost reduction using some of the rules in $\mathcal{T}_{cons}$. It seems that we can express this directly in equations. That is, we could add the equation

$$\$conc(\$nil(), a) = a \qquad \bigstar$$

to this specification. However, the rule *\$conc0* in $\mathcal{T}_{cons}$ would then reduce to $[\$conc0]\ L = L$.

Now that we know how to represent lists and have functions to build lists from sublists, we will look at some examples of transformed TRSs.

12.2 Example Transformations

We look at the transformation of three TRSs that cover the kinds of list patterns that can be written in RN2. Several possibilities for the transformation have been considered. The advantage of the scheme described here is that no garbage is produced by the simulation of list matching.

12.2.1 Head/Tail Patterns

A straightforward use of associative lists is using head/tail patterns like $[X, X*]$ to divide a list in its head X and tail $[X*]$. These patterns can be translated directly into $\mathit{\$cons}/\mathit{\$nil}$ patterns. For instance, consider the TRS

```
signature sum(lstTp)
equations [sum] sum([]) = zero()
          [sum] sum([N,N*]) = plus(N,sum([N*]))
```

which sums up the naturals in a list of successor naturals. It is directly transformed to

```
signature sum(appTp)
equations [sum] sum($nil()) = zero()
          [sum] sum($cons(N,L)) = plus(N,sum(L))
```

Note that the declaration of sum in the signature is changed to $\mathit{sum}(\mathit{appTp})$. Lists represented by $\mathit{\$cons}$ and $\mathit{\$nil}$ are applications like applications of any other function.

12.2.2 Last

The pattern $[X*, X]$ divides the list in its initial segment $[X*]$ and its last element X. Such patterns are not directly expressable by $\mathit{\$cons}/\mathit{\$nil}$ patterns since the prefix $[X*]$ may contain any number of elements. We define, for each n a function $\mathit{\$lastn}$, that extracts the last n elements from a $\mathit{\$cons}/\mathit{\$nil}$ list. Only the functions actually needed in a transformed TRS are generated.

As an example of this list pattern, consider the specification of the function revers that reverses the elements of a list.

```
signature revers(lstTp)
equations [rev0] revers([]) = []
          [revn] [X*,X] := L1, [Y*] := revers([X*])
          ===> revers(L1) = [X,Y*]
```

With $\mathit{\$last1}$ this TRS is transformed to

```
signature revers(appTp)
equations [rev0] revers($nil) = $nil
          [revn] L2 := $last1(L1),
                 $cons(X,$nil) := L2,
                 L3 := revers($until(L2,L1)),
                 ===> revers(L1) = $cons(X,L3)
```

The first condition yields the list $L2$, with length 1, containing the last element of the list $L1$. The second condition extracts that element X from $L2$ by matching it with the pattern $\mathit{\$cons}(X, \mathit{\$nil})$. The initial segment $[X*]$ of the list $L1$ is the list of all elements of $L1$ $\mathit{\$until}$ $L2$.

12.2.3 Head/Tail and Last Combined

Patterns with only one sublist variable can have at most one match. In the examples above we saw the cases where the sublist variables is either the last or first element of the pattern. This leaves patterns like $[X, X*, Y]$. For instance, consider the condition

$$[X, X*, Y] := L$$

it is transformed to the equivalent conditions

$$[X, Y*] := L, [X*, Y] := [Y*]$$

Now we have a head/tail pattern and a last pattern, which we can transform as shown above.

For non-deterministic lists (with more than one sublists variable) the same transformation is valid. The condition $[X, X*, Y*] := L$ is transformed to the conditions $[X, Z*] := L$ and $[X*, Y*] := [Z*]$. This means we only have to consider non-deterministic lists that start with a sublist variable. These transformations on conditions are carried out by transformation FL specified in the next chapter.

12.2.4 Non-Deterministic Patterns

The list patterns that cause all the complications are patterns with more than one sublist variable. These patterns have potentially more than one match.

We study the transformation of the TRS for the function *set* that we showed in Section 9.2. The left-linearized and flattened version of that TRS is

$$\begin{aligned}&\textit{signature set}(\textit{lstTp})\\&\textit{equations}\,[\textit{set}]\ [X^*,\ I,\ Y^*,\ J,\ Z^*] := L,\ I \stackrel{?}{=} J\\&\qquad\qquad ===>\ \textit{set}(L) = \textit{set}([X^*,\ I,\ Y^*,\ Z^*])\end{aligned}$$

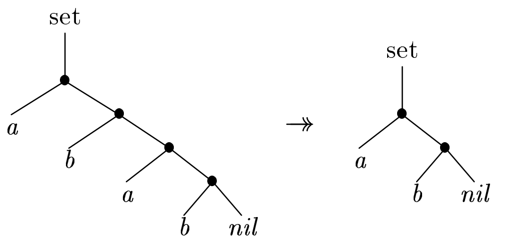
The function *set* removes duplicates from the list it is applied to. For instance,

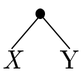
$$\textit{set}([a(),\ b(),\ a(),\ b()]) \twoheadrightarrow \textit{set}([a(),\ b()])$$

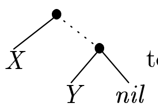
In the transformed version of this TRS, *set* applies to a *\$cons/\$nil* list. We want, therefore

$$\begin{aligned}&\textit{set}(\$\textit{cons}(a(),\ \$\textit{cons}(b(),\ \$\textit{cons}(a(),\ \$\textit{cons}(b(),\ \$\textit{nil}())))))\\&\twoheadrightarrow\\&\textit{set}(\$\textit{cons}(a(),\ \$\textit{cons}(b(),\ \$\textit{nil}())))\end{aligned}$$

Terms as Trees In the explanation below we will sometimes draw the trees by which terms are represented. For instance, the reduction above can also be drawn as



where 

represents $\$cons(X, Y)$. We will draw 

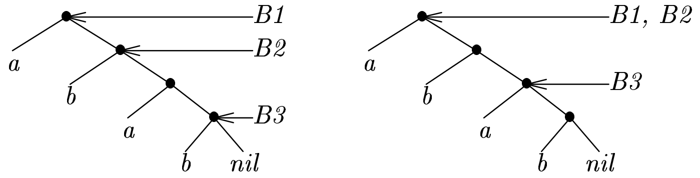
to denote the term $\$cons(X, \ldots \$cons(Y, \$nil) \ldots)$. The dots denote $n \geq 0$ terms. If $n = 0$ X and Y are the same element.

Blocks and Sublists We will call a list pattern $[\vec{a}, X*]$, where $\vec{a}$ contains no sublist variables, a *block*. Each non-deterministic list pattern can be divided into a list of blocks. The blocks for the pattern $[X*, I, Y*, J, Z*]$ are $[X*]$, $[I, Y*]$ and $[J, Z*]$. Blocks are representable as *$cons/$nil* patterns since they are in head/tail form.

The new rule for *set* must find the first division in three sublists of the list L such that $I \stackrel{?}{=} J$. That is, the first element of the second sublist is equal to the first element of the third sublist.

It may be the case that a division in sublists is made, but that the condition $I \stackrel{?}{=} J$ can not be validated for that division. In that case a new division has to be made and the condition has to be tested for the new division. This process is repeated until a division is found for which the condition holds or until no more divisions can be made.

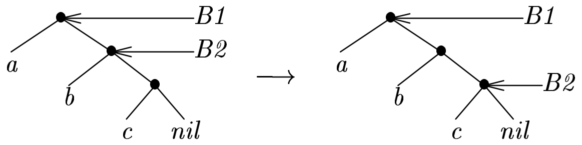
Representing Sublists We will represent the sublists of a list assigned to the blocks for a pattern by 'pointers' in the list. Where 'pointer' denotes some tail of the original list. For instance, the trees



show two divisions in sublists of the list $[a, b, a, b]$. The variables $B1$, $B2$ and $B3$ point at the start of, respectively, the first, second and third sublist. A sublist ends where the next sublist begins. The last sublist is the list at the last pointer. Thus, in the first tree the sublists are *$cons*$(a,$ *$nil*$)$, *$cons*$(b,$ *$cons*$(a,$ *$nil*$))$ and *$cons*$(b,$ *$nil*$)$. In the second tree the sublists are *$nil*, *$cons*$(a,$ *$cons*$(b,$ *$nil*$))$ and *$cons*$(a,$ *$cons*$(b,$ *$nil*$))$. Given the start and end pointers, X and Y, of a sublist the sublist is constructed by *$until*$(Y, X)$. We repeat the rules for *$until* from TRS $\mathcal{T}_{cons}$ (Section 12.1).

```
[$unt0] $until($nil(),L2) = L2
[$unt0] $eq(L1,L2) ?= yes() ===> $until(A1,A2) = $nil()
[$untn] $eq(L1,L2) ?= no(), $cons(I,L3) := L2
        ===>
        $until(L1,L2) = $cons(I,$until(L1,L3))
```

By putting a pointer one step further in the list, the sublist it ends gets one more element and the sublist it starts gets one less. For instance, the transition



means that the first sublist goes from $[a]$ to $[a, b]$ and the second from $[b, c]$ to [c].

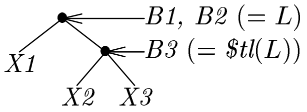
The New Rule for Set The declaration of *set* is changed to

$$set(appTp)$$

The rule for *set* is transformed to

```
[set] $cons(X1,$cons(X2,X3)) := L,
      L'                     := $try(L, L, $tl(L), L),
      L'                     ≠ $fail()
      ===> set(L) = L'
```

The first condition checks if the list L contains at least two elements. If that is the case, the function *$try* is called with the initial division in sublists



This means that the first division in sublists is *$nil*, *$cons*$(X1,$ *$nil*) and *$cons*$(X2, X3)$.

The result of the application of *$try* is assigned to L'. If L' is unequal to *$fail*(), L' is the reduct of *set*(L). Otherwise the rule fails and *set*(L) is in normal form (assuming there are no other rules for *set*).

The Function Try tests if a division in sublist validates the condition $I \stackrel{?}{=} J$. If that is not the case the function *$next* is called that makes a new division. The declaration of *$try* is

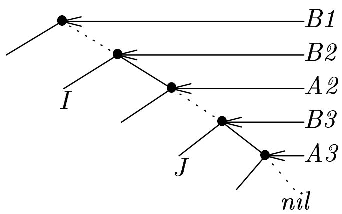
$$\$try(appTp,\ appTp,\ appTp,\ appTp)$$

The first three arguments are the blocks. The last argument is the variable L which might be used in one of the conditions or the rhs (which is not the case here).

There are two rules for *$try*. The first rule tries to validate the condition $I \stackrel{?}{=} J$. If that succeeds, the rhs for the rule is constructed. The rule is

```
[$try] $cons(I, A2) := B2, $cons(J, A3) := B3, I ?= J
       ===> $try(B1, B2, B3, L)
       = set($concUntil(B2, B1, $cons(I, $concUntil(B3, A2, A3))))
```

The first and second condition assign values to I and J. Then the second condition of the original rule is checked ($I \stackrel{?}{=} J$). If that succeeds the rule for *set* applies. We now have the following pointers in the list.



With this information the rhs has to be constructed. We need the sublists that correspond with the sublist variables in the rhs. The sublists $[X*]$, $[Y*]$ and $[Z*]$ are *$until*$(B2, B1)$, *$until*$(B3, A2)$ and $A3$. The original rhs was *set*$([X*, I, Y*, Z*])$. A first translation to cons/nil notation is

$$set(\$conc([X*], \$cons(I, \$conc([Y*], \$conc([Z*], \$nil)))))$$

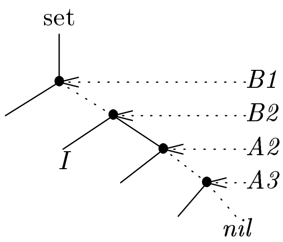
After substituting the sublists by their new values we get

$$set(\$conc(\$until(B2, B1), \$cons(I, \$conc(\$until(B3, A2), \$conc(A3, \$nil)))))$$

Optimizing this with *fold-cn* gives us

$$set(\$concUntil(B2, B1, \$cons(I, \$concUntil(B3, A2, A3))))$$

This will reduce to the term



where $B1$, $B2$, $A2$ and $A3$ show the locations in the list that correspond with the locations in the original list.

If validation of the conditions in the previous rule did not succeed the next configuration is made by *\$next*.

$$\begin{aligned} \textit{otherwise: } [\$try]\ & \$try(B1,\ B2,\ B3,\ E1) \\ & = \$next(B1,\ B2,\ B3,\ B1,\ B2,\ B3,\ E1) \end{aligned}$$

The Function Next Given a division in sublists, for which *\$try* failed to validate the conditions, the function *\$next* finds the next division. The declaration of *\$next* is

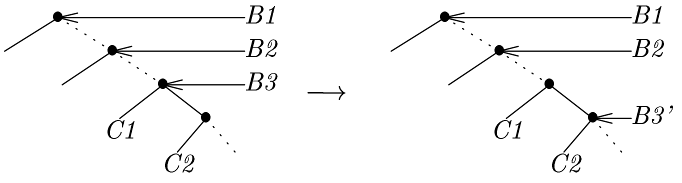
$$\$next(appTp,\ appTp,\ appTp,\ appTp,\ appTp,\ appTp,\ appTp)$$

The first three arguments serve to analyze the division and decide which step to take. The fourth through sixth argument are used to build the new division. The last argument is the environment variable L. This duplication of arguments has the advantage that the list is not reconstructed at the rhs which produces unnecessary garbage.

There are three cases to deal with. We label the cases by the sublist they deal with and present them in reverse.

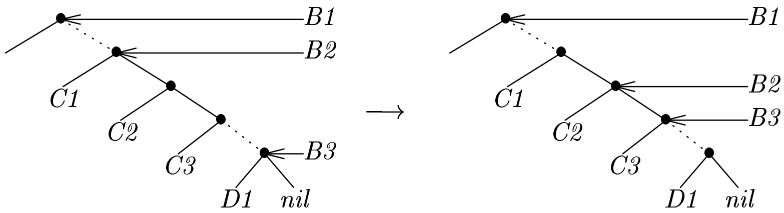
Sublist 3 If the last sublist contains more than one element, we shift an element from the last to the second sublist.

$$\begin{aligned} [\$next]\ & \$next(A1,\ A2,\ \$cons(C1,\ \$cons(C2,\ C\$)),\ B1,\ B2,\ B3,\ E1) \\ & = \$try(B1,\ B2,\ \$tl(B3),\ E1) \end{aligned}$$



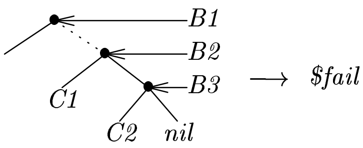
Sublist 2 If the last sublist sublist contains one element and the second sublist contains more than one element (which means that the list starting at $B2$ has more than two elements, since it includes the one element of the last sublist) one element is shifted from the second to the first sublist and the last sublist is reinitialized.

```
[$next] B2' := $tl(B2), B3' := $tl(B2')
        ===>
        $next(A1, $cons(C1, $cons(C2, $cons(C3, C$))),
                  $cons(D1, $nil()),
                  B1, B2, B3, E1)
        = $try(B1, B2', B3', E1)
```

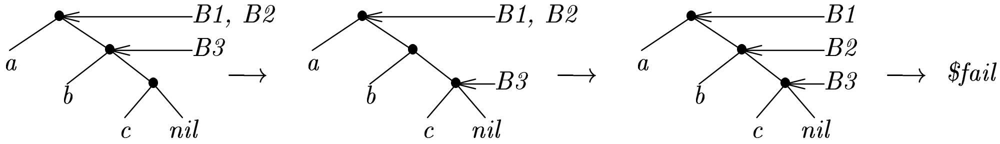


Sublist 1 If the list starting at $B2$ has exactly two elements. No element can be shifted from the second or third sublist since the second and third sublist should have at least one element each. Since this division was already tested by *$try*, no more possibilities for a match exist. The failure is reported by the constant *$fail*(). The third condition in the rule for *set* will then fail. This means that there were no duplicates found in the list.

```
[$next] $next(A1, $cons(C1, $cons(C2, $nil())), A3, B1, B2, B3, E1) = $fail()
```



Example We show all divisions that will be produced by next for the list *$cons*($a$, *$cons*(b, *$cons*($c$, *$nil*)))



Example A division in sublists can also be depicted as $|\ldots|\ldots|$. The function next makes the following divisions for the list $[a, b, c, d]$.

$$||a|bcd| \to ||ab|cd| \to ||abc|d| \to |a|b|cd| \to |a|bc|d| \to |ab|c|d| \to \textit{\$fail()}$$

We have now seen three examples of transformed TRSs. We saw in these examples that there are several classes of patterns that get a different treatment by the transformation. We start the definition of the transformation in the next section with a classification of list patterns.

12.3 Classification of List Expressions

Imports Environments$^{(9.5)}$

We distinguish 8 types of list patterns. Each type gets a different treatment by the transformations *fl* and *lm*. The types are split in two main groups; deterministic list patterns have at most one match and non-deterministic list patterns can have more than one match. Table 12.1 shows the classification.

The patterns we saw in the examples of the previous section have the following types.

$$
\begin{array}{lcl}
[] & \mapsto & 1 \\
[X, X*] & \mapsto & 4 \\
[X*, X] & \mapsto & 3 \\
[X*, I, Y*, J, Z*] & \mapsto & 8
\end{array}
$$

We saw that patterns like $[X, X*, Y]$ can be translated to a pattern $[X, Y*]$ of type 4 and a pattern $[X*, Y]$ of type 3. Similarly, patterns of type 5 (6) can be translated to a pattern of type 4 and a pattern of type 7 (8). These transformations are carried out by transformation FL. Therefore, in transformation LM, we do not have to deal with patterns of type 2, 5 and 6.

Type	Description[a]	*count slVar?*	*slVar?* $\circ$ *hd*	*slVar?* $\circ$ *last*
	Deterministic lists			
1	$[\vec{c}]$	0	–	–
2	$[\vec{a}, x^*, \vec{b}]$	1	$\bot$	$\bot$
3	$[x^*, \vec{b}]$	1	$\top$	$\bot$
4	$[\vec{a}, x^*]$	1	–	$\top$
	Non-deterministic lists			
5	$[\vec{a}, x^*, \vec{c}, y^*, \vec{b}]$	> 1	$\bot$	$\bot$
6	$[\vec{a}, x^*, \vec{c}, y^*]$	> 1	$\bot$	$\top$
7	$[x^*, \vec{c}, y^*, \vec{b}]$	> 1	$\top$	$\bot$
8	$[x^*, \vec{c}, y^*]$	> 1	$\top$	$\top$

[a] $\vec{a}$, $\vec{b}$ and $\vec{c}$ are lists of TERMs, where $\vec{a}$ and $\vec{b}$ have length > 1 and no sublist variables and $\vec{c}$ has length ≥ 0 and possibly sublist variables.

Table 12.1: Classification table of list expressions.

Exports

Context-free Functions

list-kind $\rightarrow$ Combinator
lk-table $\rightarrow$ Combinator

Equations

Table 12.1 is encoded in the definition of the function *list-kind*.

$$
\frac{\begin{array}{l}
n \; = \textit{count slVar?}\; [\![\vec{t}\,]\!]_{\vec{t}}, \\
\mathcal{B}_1 = \neg\,(\varepsilon?\; [\![\vec{t}\,]\!]_{\vec{t}}) \rightarrow \textit{slVar?}\; (\textit{hd}\; [\![\vec{t}\,]\!]_{\vec{t}}), \\
\mathcal{B}_2 = \neg\,(\varepsilon?\; [\![\vec{t}\,]\!]_{\vec{t}}) \rightarrow \textit{slVar?}\; (\textit{last}\; [\![\vec{t}\,]\!]_{\vec{t}})
\end{array}}{\textit{list-kind}\; [\![\vec{t}\,]\!]_{\vec{t}} = \textit{lk-table}\; \langle n, \mathcal{B}_1, \mathcal{B}_2\rangle}
$$

The case analysis is done by the combinator *lk-table*.

$$
\begin{array}{ll}
\textit{lk-table}\; \langle 0, \mathcal{B}_1, \mathcal{B}_2\rangle = 1 & \textit{lk-table}\; \langle n, \bot, \bot\rangle = 5 \Leftarrow n > 1 = \top \\
\textit{lk-table}\; \langle 1, \bot, \bot\rangle \;\; = 2 & \textit{lk-table}\; \langle n, \bot, \top\rangle = 6 \Leftarrow n > 1 = \top \\
\textit{lk-table}\; \langle 1, \top, \bot\rangle \;\; = 3 & \textit{lk-table}\; \langle n, \top, \bot\rangle = 7 \Leftarrow n > 1 = \top \\
\textit{lk-table}\; \langle 1, \mathcal{B}_1, \top\rangle \;\; = 4 & \textit{lk-table}\; \langle n, \top, \top\rangle = 8 \Leftarrow n > 1 = \top
\end{array}
$$

Henceforth we shall refer to list patterns by their type. The next section deals with the division of list patterns in blocks.

12.4 Blocks

Imports RN2-Terminology$^{(9.4)}$

In the example of the transformation of the rule for *set* we saw how a non-deterministic list can be split into blocks, where a block is a list pattern of type 1 or 4. In matching a non-deterministic pattern against a list, we must divide the list in sublists, one for each block.

Here, we define the combinator *blocks* that translates a list of terms to a triple of the number of constant terms (non-sublist variables), the number of blocks and the list of blocks.

Exports
Context-free Functions
blocks → Combinator {**type:** TERM∗ → $\mathbb{N} \times \mathbb{N} \times$ BLOCK∗}

where the type BLOCK is defined as

$$\text{BLOCK} = \text{TERM}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{B}$$

A block contains a list of terms (a pattern of type 1 or 4), two naturals and a boolean. These last three values are, respectively, the number of constant terms in the block, the number of constant terms in the block plus the number of constant terms in the blocks following the block and a boolean that is true when the last element of the block is a sublist variable, false otherwise.

To access the datastructure generated by *blocks* the following combinators are defined.

Exports
Context-free Functions
"$block_i$" → Combinator {**type:** $\mathbb{N} \times \mathbb{N} \times$ BLOCK∗ → BLOCK}
"$block_{\#c}$" → Combinator {**type:** $\mathbb{N} \times \mathbb{N} \times$ BLOCK∗ → $\mathbb{N}$}
"$block_{\#}$" → Combinator {**type:** $\mathbb{N} \times \mathbb{N} \times$ BLOCK∗ → $\mathbb{N}$}

"$block_{terms}$" → Combinator {**type:** BLOCK → TERM∗}
"$block_{consts}$" → Combinator {**type:** BLOCK → $\mathbb{N}$}
"$block_{acc\text{-}consts}$" → Combinator {**type:** BLOCK → $\mathbb{N}$}
"$block_{last}$" → Combinator {**type:** BLOCK → $\mathbb{B}$}

Exports
Variables
"$\mathfrak{B}_{DS}$" [0-9′]∗ → Term
"$\mathfrak{B}$" [0-9′]∗ → Term
"$\vec{\mathfrak{B}}$" [0-9′]∗ → {Term "," }∗

Hiddens
Context-free Functions
add → Combinator

Equations

The division of a list pattern in blocks is done by *blocks*.

$$blocks\ [\![\vec{t}]\!]_{\vec{t}} = foldr\ add\ \langle 0,\, 0,\, []\rangle\ [\![\vec{t}]\!]_{\vec{t}}$$

The datastructure is built starting at the last element of the pattern. Each element is added to the datastructure by *add*. If the term to be added is not a sublist variable and is the last element of the pattern, a new block is added to the list of blocks.

$$add\ t\ \langle 0,\, 0,\, []\rangle = \langle 1,\, 1,\, [\langle [\![t]\!]_{\vec{t}},\, 1,\, 1,\, \bot\rangle]\rangle \Leftarrow type\ t \neq slTp$$

If the term is a sublist variable, a new block is created. The block has 0 constant terms and the blocks in the rest of the pattern have j constant terms.

$$add\ x^*\ \langle i,\, j,\, [\vec{\mathfrak{B}}]\rangle = \langle i+1,\, j,\, [\langle [\![x^*]\!]_{\vec{t}},\, 0,\, j,\, \top\rangle,\, \vec{\mathfrak{B}}]\rangle$$

If the term is not a sublist variable and not the last element of the pattern the term is added to the first block in the list of blocks. The naturals k and m, the number of constants in the block itself and in all blocks, respectively, are increased by one.

$$\frac{type\ t \neq slTp}{add\ t\ \langle i,\, j,\, [\langle [\![\vec{t}]\!]_{\vec{t}},\, k,\, m,\, \mathcal{B}\rangle,\, \vec{\mathfrak{B}}]\rangle = \langle i,\, j+1,\, [\langle [\![t,\, \vec{t}]\!]_{\vec{t}},\, k+1,\, m+1,\, \mathcal{B}\rangle,\, \vec{\mathfrak{B}}]\rangle}$$

Example For the pattern $[X*, I, Y*, J, Z*]$ *blocks* yields

$$\langle 3, 2, [\langle [\![X*]\!]_{\vec{t}}, 0, 2, \top\rangle, \langle [\![I, Y*]\!]_{\vec{t}}, 1, 2, \top\rangle, \langle [\![J, Z*]\!]_{\vec{t}}, 1, 1, \top\rangle]\rangle$$

The Projection Functions

$$block_i\ i\ \langle k,\, m,\, [\vec{\mathfrak{B}}]\rangle = \pi\ i\ [\vec{\mathfrak{B}}] \qquad block_{\#c}\ \langle k,\, m,\, [\vec{\mathfrak{B}}]\rangle = m \qquad block_{\#}\ \langle k,\, m,\, [\vec{\mathfrak{B}}]\rangle = k$$

$$block_{terms} = \pi\ 1 \qquad block_{consts} = \pi\ 2 \qquad block_{acc\text{-}consts} = \pi\ 3 \qquad block_{last} = \pi\ 4$$

12.5 The Function Last

Imports Representing-Lists-by-Cons-and-Nil$^{(12.1)}$

Now we specify the function *gen-last* that, given a list of naturals, generates a TRS that contains, for each natural n in that list, the declaration and rules for the function *\$lastn*. The function *\$lastn* yields the last n elements of the list it is applied to, if the list contains at least n elements.

Exports

Context-free Functions

"*Last*" $\rightarrow$ Combinator {**type:** $\mathbb{N} \rightarrow$ TERM $\rightarrow$ TERM}
gen-last-i $\rightarrow$ Combinator {**type:** $\mathbb{N} \rightarrow$ TRS}
gen-last $\rightarrow$ Combinator {**type:** $\mathbb{N}* \rightarrow$ TRS}

Equations

The combinator *Last* constructs an application of *\$lastn* to t.

$$Last\ n\ t = app\ (\$last\ \_\ n)\ [\![t]\!]_{\vec{t}}$$

Given a natural n *gen-last-i* generates a TRS containing the function declaration and rules for *\$lastn*. The function takes the tail of a list until the tail contains precisely n elements. The first rule returns the list if it contains exactly n elements. The second rule recursively applies *\$lastn* to the tail of the list, if it contains more than n elements.

$$\frac{\begin{array}{l} a = cons_t\ \mathbf{I}\ (list\ (A\ \_\ (1\ ..\ n))), \\ b = cons_t\ \mathbf{I}\ (Conc\ (list\ (A\ \_\ (1\ ..\ (n+1))))\ B), \\ f = \$last\ \_\ n \end{array}}{\begin{array}{rl} gen\text{-}last\text{-}i\ n = & signature\ f(appTp) \\ & equations\ [f]\ a := L ===> f(L) = L \\ & \qquad\qquad\quad [f]\ b := L ===> f(L) = f(\$tl(L)) \end{array}}$$

For instance, for $n = 1$ *gen-last-i* generates

$$\begin{array}{ll} \textit{signature} & \textit{\$last1}(\textit{appTp}) \\ \textit{equations} & [\textit{\$last1}]\ \textit{\$cons}(A1,\textit{\$nil}())\ := L ===> \textit{\$last1}(L)\ = L \\ & [\textit{\$last1}]\ \textit{\$cons}(A1,\textit{\$cons}(A2,B))\ := L \\ & \qquad ===> \textit{\$last1}(L)\ = \textit{\$last1}(\textit{\$tl}(L)) \end{array}$$

Note that we could have solved this by making one generic function that has n as an argument. However this implies that the transformed TRS must contain natural numbers. Furthermore it would cost many more reduction steps to check whether the list has n or more elements than with this solution, where the check is carried out by the reduction machine. The number of *$lastn* functions will in practice not be large.

Finaly, combinator *gen-last*, given a list of naturals, generates for each n in the list the corresponding specification of the *$lastn* function and unites the specifications with $+$.

$$\begin{array}{rcl} \textit{gen-last}\,[\vec{n}] & = & \textit{foldr}\,(+)\ \textit{signature}\ \ \textit{equations} \\ & & ((\textit{map gen-last-i} \circ \textit{unique} \circ \textit{sort}\,(\leq))\,[\vec{n}]) \end{array}$$

Note that the list of naturals is sorted and that duplicates are removed before the functions are generated. For each n in the list, exactly one function declaration and two rules are generated.

We now have all ingredients to define the transformation, except for the generation of the functions *$try* and *$next* for a non-deterministic list pattern. We will first define the transformation and then, in Section 12.7, define the combinator *gen-try-next* that generates the rules for *$try* and *$next*.

12.6 Transformation LM

Imports Classification-of-List-Expressions$^{(12.3)}$ Non-Deterministic-Lists$^{(12.7)}$

The transformation *lm*, which transforms a flat TRS into a CD-RN2 TRS, in wich associative lists are represented by *$cons*/*$nil* lists, consists of the functions

Exports

Context-free Functions

"*lm*" "⟦" TRS "⟧" → TRS
"*lm*" "⟦" TERM "⟧" → TERM
"lm^{-1}" "⟦" TERM "⟧" → TERM

The pre- and postcondition are

$$lm_{\mathrm{pre}}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{RN2}_{\mathit{fl}} \qquad lm_{\mathrm{post}}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{CDRN2} \qquad \bigstar$$

The transformation has three main tasks. The *lstTp* arguments in function declarations are changed to *appTp*. Instantiating terms are translated with $cons_t$ to *$cons*/*$nil* notation. Matching terms without lists or matching terms with only (deterministic) lists of type 1 or 4 are also translated with $cons_t$. Assignment conditions with lhs's that are lists of type 3, 7 or 8 are replaced with new conditions that call auxiliary functions to carry out the list matching for the pattern. The subterms of such lists are transformed with $cons_t$.

The following combinators are used in the definition of the transformation.

Exports

Context-free Functions

"$lm_{\mathcal{T}}$" → Combinator {**type:** TRS → TRS}
"lm_{fd}" → Combinator {**type:** FUNDECL → $\mathbb{E} \to \mathbb{E}$}
"lm_{φ}" → Combinator {**type:** RULE → $\mathbb{E} \to \mathbb{E}$}
"lm_{C}" → Combinator {**type:** $\mathrm{COND}^* \to \mathbb{E} \to \mathbb{E}$}

Equations

The transformation

$$lm[\![\mathcal{T}]\!] = \mathcal{T}' \Leftarrow \mathcal{T}' = lm_{\mathcal{T}}\ \mathcal{T}$$

Ground terms over LRN2 are translated to terms over BRN2 by $cons_t$ and back by $cons_t^{-1}$.

$$lm[\![t]\!] = t' \Leftarrow t' = cons_t\ \mathbf{I}\ t \qquad lm^{-1}[\![t]\!] = t' \Leftarrow t' = cons_t^{-1}\ t$$

TRSs Function declarations and rules are transformed. The resulting TRS is concatenated with the specification $\mathcal{T}_{cons}$ and the specification for *\$last* functions needed for this TRS.

$$\frac{\mathcal{E} = foldr\ lm_{\varphi}\ (foldr\ lm_{fd}\ (set_{\rho}\ (\Delta_{\rho}\ (\backslash *))\ (new\text{-}env\ \$lm\ \text{LM}))\ (\Sigma\ \mathcal{T}))\ (\mathbf{R}\ \mathcal{T})}{lm_{\mathcal{T}}\ \mathcal{T} = \mathcal{T}_{cons} + gen\text{-}last\ (\pi_{last}\ \mathcal{E}) + trs\ (\pi_{\vec{fd}}\ \mathcal{E})\ (\pi_{\vec{\varphi}}\ \mathcal{E})}$$

Function declarations Since terms of type *lstTp* are transformed to terms of type *appTp* we translate all function declarations $f(\ldots, lstTp, \ldots)$ to $f(\ldots, appTp, \ldots)$.

$$lm_{fd}\ f(\vec{\tau})\ \mathcal{E} = fun\text{-}decl\ f\ (map\ (\mathbf{K}\ appTp)\ [\![\vec{\tau}]\!]_{\vec{\tau}}) :_{fd} \mathcal{E}$$

Rules The lhs is transformed to cons notation by $cons_t$, the conditions are transformed by lm_C. The *vars* field in the environment keeps track of the variables introduced.

$$\frac{\begin{array}{l} t \ \ = cons_t\ (\pi_{\rho}\ \mathcal{E}_1)\ (lhs\ \varphi), \\ \mathcal{E}_2 = (lm_C\ (conds\ \varphi) \circ set_{vars}\ (vars\ t) \circ set_{\vec{C}}\ [\![]\!]_{\vec{C}} \circ set_{rhs}\ (rhs\ \varphi))\ \mathcal{E}_1 \end{array}}{lm_{\varphi}\ \varphi\ \mathcal{E}_1 = rule\ (kind\ \varphi)\ (tag\ \varphi)\ (\pi_{\vec{C}}\ \mathcal{E}_2)\ t\ (\pi_{rhs}\ \mathcal{E}_2) :_{\varphi} \mathcal{E}_2}$$

Conditions If no more conditions are left we are ready with the rule. The rhs is transformed using the assignment to sublist variables that is build during the transformation. The assignment in the environment is set to the default assignment $(\backslash *)$, that unstars sublist variables.

$$lm_C\ [\![]\!]_{\vec{C}}\ \mathcal{E} = set_{\rho}\ (\Delta_{\rho}\ (\backslash *))\ (set_{rhs}\ (cons_t\ (\pi_{\rho}\ \mathcal{E})\ (\pi_{rhs}\ \mathcal{E}))\ \mathcal{E})$$

If C is a positive or negative condition the lists in the lhs and rhs are transformed into *\$cons*-lists.

$$\frac{new?\ C = \bot}{\begin{array}{l} lm_C\ [\![C, \vec{C}]\!]_{\vec{C}}\ \mathcal{E} = cond\ (kind\ C)\ (cons_t\ (\pi_{\rho}\ \mathcal{E})\ (lhs\ C))\ (cons_t\ (\pi_{\rho}\ \mathcal{E})\ (rhs\ C)) \\ \qquad\qquad\qquad\quad :_C lm_C\ [\![\vec{C}]\!]_{\vec{C}}\ \mathcal{E} \end{array}}$$

If C is a new condition, but the lhs is not a list or a list of type 1 or 4, the lhs and rhs are translated to cons notation and the variables of the lhs are added to the environment.

$$\frac{\begin{array}{ll} \neg\ (list?\ s) \vee (list\text{-}kind\ (items\ s) \in [1, 4]) = \top, & \\ s' & = cons_t\ (\pi_{\rho}\ \mathcal{E})\ s, \\ t' & = cons_t\ (\pi_{\rho}\ \mathcal{E})\ t \end{array}}{lm_C\ [\![s := t, \vec{C}]\!]_{\vec{C}}\ \mathcal{E} = s' := t' :_C lm_C\ [\![\vec{C}]\!]_{\vec{C}}\ (vars\ s' +\!\!+_{vars}\ \mathcal{E})}$$

Remain the cases where the lhs of the assigment is a list of type 3, 7 or 8.

Lists of Type 3 have the form $[x^*, \vec{s}]$, where $\vec{s}$ is not empty and contains no sublist variables. Let n be the length of $\vec{s}$. The suffix of the list that is assigned to this pattern is first retrieved with the function *$lastn*, which takes the tail of the list until exactly n elements are left. The resulting list is assigned to a new variable x. The variable is assigned to the transformation to cons notation of $[\vec{s}]$. Everywhere in the rest of the rule, x^* is replaced with *$until*$(x, t)$, where t is the rhs of the original condition.

The natural n is added to the *last* list of the environment. For each i in that list the definition of the function *$lasti* is generated by *gen-last*. The generation of the *$last* functions will be defined in module 12.5.

$$\frac{\begin{array}{ll} \textit{list-kind}\ [\![x^*,\ \vec{s}]\!]_{\vec{t}} = 3, & \\ n & = \#\ [\![\vec{s}]\!]_{\vec{t}}, \\ \langle \mathcal{E}_2,\ x\rangle & = \textit{alloc-var}\ \mathcal{E}_1, \\ s & = \textit{cons}_t\ \mathbf{I}\ [\vec{s}], \\ t' & = \textit{cons}_t\ (\backslash *)\ t, \\ t'' & = \textit{Last}\ n\ t' \end{array}}{\begin{array}{rl} \textit{lm}_C\ [\![[x^*,\ \vec{s}] := t,\ \vec{C}]\!]_{\vec{C}}\ \mathcal{E}_1 = & (([\![x := t'',\ s := x]\!]_{\vec{C}}\ +\!\!+_{\vec{C}}) \\ & \circ\ \textit{lm}_C\ [\![\vec{C}]\!]_{\vec{C}} \\ & \circ\ (n :_{last}) \\ & \circ\ (x : \textit{vars}\ s\ +\!\!+_{\textit{vars}}) \\ & \circ\ \textit{assign}\ (x^* \mapsto \textit{\$until}(x,\ t')))\ \mathcal{E}_2 \end{array}}$$

For instance, the rule

$$[\textit{paln}]\ [A*, A] := [B*], A \stackrel{?}{=} B ===> \textit{palindrome}([B, B*]) = \textit{palindrome}([A*]) \qquad \bigstar$$

is transformed to

$$\begin{array}{ll} [\textit{paln}] & Y := \textit{\$last1}(B\$), \textit{\$cons}(A, \textit{\$nil}()) := Y, A \stackrel{?}{=} B \\ & ===> \textit{palindrome}(\textit{\$cons}(B, B\$)) = \textit{palindrome}(\textit{\$until}(Y, B\$)) \end{array} \qquad \bigstar$$

Lists of Type 7 and 8 are lists of the form $[x^*, \vec{t}]$, where $\vec{t}$ is not empty and contains at least one sublist variable. These lists have potentially more than one match. In module 11.3 we saw that the matching function *match* handles sublist variables different than normal variables. Since we are transforming specifications to specifications without lists and thus without sublists variables, the matching of non-deterministic patterns has to be handled in the transformed specification. Therefore we generate for each pattern of type 7 or 8 two functions that examine all possible matches in turn, trying to validate the rest of the conditions. These functions are generated by the combinator *gen-try-next*. The generation is defined in the next section.

For now it suffices to know that *gen-try-next* produces new conditions $\vec{C}_2$, that replace the old conditions $\vec{C}_1$, and a new rhs a, for the rule being transformed. The declarations and rules for the new functions are added to the environment.

$$\frac{\begin{array}{ll} \textit{list-kind}\ [\![\vec{s}]\!]_{\vec{t}} \in [7,\ 8] = \top, & \\ \langle [\![\vec{C}_2]\!]_{\vec{C}},\ a,\ \mathcal{E}_2\rangle & = \textit{gen-try-next}\ (\textit{blocks}\ [\![\vec{s}]\!]_{\vec{t}})\ t\ [\![\vec{C}_1]\!]_{\vec{C}}\ (\pi_{rhs}\ \mathcal{E}_1)\ (\pi_{vars}\ \mathcal{E}_1)\ \mathcal{E}_1 \end{array}}{\textit{lm}_C\ [\![[\vec{s}] := t,\ \vec{C}_1]\!]_{\vec{C}}\ \mathcal{E}_1 = \textit{set}_{\vec{C}}\ [\![\vec{C}_2]\!]_{\vec{C}}\ (\textit{set}_{rhs}\ a\ \mathcal{E}_2)}$$

12.7 Non-Deterministic Lists

Imports Blocks$^{(12.4)}$ The-Function-Last$^{(12.5)}$ Transformation-LM$^{(12.6)}$

The function *gen-try-next* generates the new conditions and rhs for the rule at hand and generates function declarations and rules for the functions *next* and *try* for the list pattern being handled.

Exports

Context-free Functions

gen-try-next → Combinator {**type:** $\mathbb{N} \times \mathbb{N} \times$ BLOCK* → TERM → COND*}
→ TERM → VAR* → $\mathbb{E}$
→ COND* × TERM × $\mathbb{E}$

In the definition of *gen-try-next* the following auxiliary functions are used.

Hiddens

Context-free Functions

"*ListN*" → Combinator {**type:** $\mathbb{N}$ → APVAR → TERM}
"*ListN*$^+$" → Combinator {**type:** $\mathbb{N}$ → APVAR → TERM}

new-conds → Combinator {**type:** FUNSYM → $\mathbb{N} \times \mathbb{N} \times$ BLOCK* → TERM}
→ VAR* → $\mathbb{E}$ → COND* × APVAR × $\mathbb{E}$
gen-crhs → Combinator {**type:** TERM → $\mathbb{N}$ → BLOCK → $\mathbb{N}$ × TERM}
gen-crhs-terms → Combinator {**type:** TERM → BLOCK* →}

gen-try → Combinator {**type:** FUNSYM → $\mathbb{N} \times \mathbb{N} \times$ BLOCK* → COND* }
→ TERM → VAR* → VAR* → VAR* → $\mathbb{E}$ → $\mathbb{E}$
gen-conds → Combinator {**type:** VAR* → VAR* → BLOCK*}
→ COND* × Assign
gen-cond → Combinator {**type:** VAR × VAR × VAR × TERM* }
→ COND* × Assign → COND* × Assign

gen-try-fail → Combinator {**type:** FUNSYM → FUNSYM → VAR*}
→ VAR* → RULE

gen-next → Combinator {**type:** FUNSYM → FUNSYM → $\mathbb{N} \times \mathbb{N} \times$ BLOCK* }
→ VAR* → VAR* → VAR* → VAR* → TRS
pre-next → Combinator {**type:** $\mathbb{N} \times \mathbb{N} \times$ BLOCK* → $\mathbb{N} \times \mathbb{N} \times$ BLOCK*}
gen-next-i → Combinator {**type:** FUNSYM → FUNSYM → $\mathbb{N} \times \mathbb{N} \times$ BLOCK*}
→ VAR* → VAR* → VAR* → VAR* → $\mathbb{N}$
→ RULE
next-cond → Combinator {**type:** BLOCK → VAR → TERM}

We declare the FUNSYM variables *try* and *next* for better readability of the equations.

Hiddens

Variables

try → FUNSYM
next → FUNSYM

Equations

ListN and ListN-Plus A cons-list of precisely n elements.

ListN n x = *List* (*) (x _ (1 .. n))

A cons-list of n or more elements.

ListN$^+$ n x = *List* (*) (x _ (1 .. n) <: x *)

Gen-Try-Next generates the new conditions for the rule being transformed and generates the rules for two new functions *try* and *next*. We read through the conditions of the rule. First two new functions are allocated from the environment. We will indicate the new functions by the variables *try* and *next*.

Then the number of blocks k is retrieved from the block datastructure $\mathfrak{B}_{DS}$. Four lists of fresh variables $\vec{a}$, $\vec{b}$ and $\vec{c}$ are allocated. They contain a variable for each block. The list of variables $\vec{e}'$ is used to denote the environment variables.

After this initialisation the conditions and rules can be generated. *New-conds* generates the new conditions and the new rhs for the transformed rule.

Gen-next generates the TRS $\mathcal{T}$, which contains the function declaration and rules for the function *next*. *Gen-try-fail* generates the failure rule φ for *try*. The rules $(\mathbf{R}\ \mathcal{T})$ and φ and the declarations $(\Sigma\ \mathcal{T})$ are added to the environment.

Finaly *gen-try* generates the positive rule for try.

$$\frac{\begin{array}{ll}
\langle \mathcal{E}_2, [\mathrm{try}, \mathrm{next}]\rangle & = \mathit{alloc\text{-}nfuns}\ 2\ \mathcal{E}_1, \\[1ex]
k & = \mathit{block}_{\#}\ \mathfrak{B}_{DS}, \\
\langle \mathcal{E}_3, [\![\vec{a}]\!]_{\vec{t}}\rangle & = \mathit{alloc\text{-}nvars}\ k\ \mathcal{E}_2, \\
\langle \mathcal{E}_4, [\![\vec{b}]\!]_{\vec{t}}\rangle & = \mathit{alloc\text{-}nvars}\ k\ \mathcal{E}_3, \\
\langle \mathcal{E}_5, [\![\vec{c}]\!]_{\vec{t}}\rangle & = \mathit{alloc\text{-}nvars}\ k\ \mathcal{E}_4, \\
\langle \mathcal{E}_6, [\![\vec{e}\,']\!]_{\vec{t}}\rangle & = \mathit{alloc\text{-}nvars}\ (\#\ [\![\vec{e}]\!]_{\vec{t}})\ \mathcal{E}_5, \\[1ex]
\langle [\![\vec{C}_2]\!]_{\vec{C}}, s', \mathcal{E}_7\rangle & = \mathit{new\text{-}conds}\ \mathrm{try}\ \mathfrak{B}_{DS}\ s\ [\![\vec{e}]\!]_{\vec{t}}\ \mathcal{E}_6, \\[1ex]
\mathcal{T} & = \mathit{gen\text{-}next}\ \mathrm{next}\ \mathrm{try}\ \mathfrak{B}_{DS}\ [\![\vec{e}\,']\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{c}]\!]_{\vec{t}}, \\
\varphi & = \mathit{gen\text{-}try\text{-}fail}\ \mathrm{next}\ \mathrm{try}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{e}\,']\!]_{\vec{t}}, \\[1ex]
\mathcal{E}_8 & = \varphi :_{\varphi} (\mathbf{R}\ \mathcal{T} +\!\!+_{\vec{\varphi}} (\Sigma\ \mathcal{T} +\!\!+_{\vec{fd}} \mathcal{E}_7)), \\[1ex]
\mathcal{E}_9 & = \mathit{gen\text{-}try}\ \mathrm{try}\ \mathfrak{B}_{DS}\ [\![\vec{C}_1]\!]_{\vec{C}}\ t\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ \mathcal{E}_8
\end{array}}{\mathit{gen\text{-}try\text{-}next}\ \mathfrak{B}_{DS}\ s\ [\![\vec{C}_1]\!]_{\vec{C}}\ t\ [\![\vec{e}]\!]_{\vec{t}}\ \mathcal{E}_1 = \langle [\![\vec{C}_2]\!]_{\vec{C}}, s', \mathcal{E}_9\rangle}$$

New Conditions The combinator *new-conds* generates the new conditions and the new rhs for the rule being transformed. We start with setting k to the number of contstants in the pattern and $[\vec{\mathfrak{B}}]$ to the list of blocks.

The first condition C_1, assigns the list (s) to a cons list of k elements. This only succeeds if the list has at least k elements.

The second condition C_2, calls the function *try* with an initial division in sublists. The result of the trial is assigned to a new variable x. The initialization of the blocks is generated by *gen-crhs*.

The third condition C_3 requires x to be unequal to *\$fail*(). If this is the case, dividing the list in sublists and validating the conditions was succesful, and the new rhs is then x.

Finally, if the list pattern was of type 7, a *\$last* function was used to extract the last n elements from the list. This is reported by adding n to the *last* field of the environment.

$$\frac{\begin{array}{ll} k & = \mathit{block}_{\#c}\ \mathfrak{B}_{DS}, \\ [\vec{\mathfrak{B}}] & = \pi\ 3\ \mathfrak{B}_{DS}, \\ \\ \langle \mathcal{E}_2,\, y\rangle & = \mathit{alloc\text{-}var}\ \mathcal{E}_1, \\ C_1 & = \mathit{ListN}^{+}\ k\ y \mathrel{\mathbf{:=}} s, \\ \\ \langle \mathcal{E}_3,\, x\rangle & = \mathit{alloc\text{-}var}\ \mathcal{E}_2, \\ [\![\vec{b}]\!]_{\vec{t}} & = \mathit{gen\text{-}crhs\text{-}terms}\ s\ [\vec{\mathfrak{B}}], \\ C_2 & = x := \mathit{try}(\vec{b},\, \vec{e}), \\ \\ C_3 & = x \neq \mathit{\$fail}(), \\ \\ \mathfrak{B} & = \mathit{last}\ [\vec{\mathfrak{B}}], \\ n & = \mathit{block}_{consts}\ \mathfrak{B}, \\ \mathcal{E}_4 & = \boldsymbol{if}^{c}\ \langle \neg\ (\mathit{block}_{last}\ \mathfrak{B}),\, (n :_{last}),\, \mathbf{I}\rangle\ \mathcal{E}_3 \end{array}}{\mathit{new\text{-}conds}\ \mathit{try}\ \mathfrak{B}_{DS}\ s\ [\![\vec{e}]\!]_{\vec{t}}\ \mathcal{E}_1 = \langle [\![C_1,\, C_2,\, C_3]\!]_{\vec{C}},\, x,\, \mathcal{E}_4\rangle}$$

The result is a tuple containing the list of new conditions, the new rhs and the updated environment.

Gen-Crhs generates the first division in sublists. s is the rhs of the condition; the list that has to be divided. We walk through the list of blocks $[\vec{\mathfrak{B}}]$, accumulating the number of constants encountered in the blocks. For each block a term *\$tl*(… *\$tl*(s) … is generated, where the dots denote k applications of *\$tl*. The idea is that the i-th sublists starts at k elements after the begining of the list if the blocks prior to the i-th block contain k constant elements.

$$\mathit{gen\text{-}crhs\text{-}terms}\ s\ [\vec{\mathfrak{B}}] = \pi\ 2\ (\mathit{mapAcc}\ (\mathit{gen\text{-}crhs}\ s)\ [\![]\!]_{\vec{t}}\ 0\ [\vec{\mathfrak{B}}])$$

So if we are at the i-th block and the blocks prior to this block contain k elements, we generate the k-th tail of s.

$$\mathit{gen\text{-}crhs}\ s\ k\ \mathfrak{B} = \langle k+m,\, \mathit{Nth\text{-}Tl}\ k\ s\rangle \Leftarrow \mathit{block}_{last}\ \mathfrak{B} = \top,\ m = \mathit{block}_{consts}\ \mathfrak{B}$$

There is one exception to this rule. If we are at the last block and that block does not end with a sublist variable (we are dealing with a pattern of type 7), we do not generate the k-th tail but apply the function *\$lastn* to s.

$$\mathit{gen\text{-}crhs}\ s\ k\ \mathfrak{B} = \langle k,\, \mathit{Last}\ n\ s\rangle \Leftarrow \mathit{block}_{last}\ \mathfrak{B} = \bot,\ n = \mathit{block}_{consts}\ \mathfrak{B}$$

Gen-Try generates the positive rule for *try*. It first extracts the constant elements from the sublists. Then checks the conditions $\vec{C}_2$ of the original rule. If that succeeds the rhs of the original rule is yielded. The conditions that extract the constant elements are generated by *gen-conds*. Furthermore, the declaration for *try* is generated.

$$\begin{array}{ll}
[\vec{\mathfrak{B}}] & = \pi\ 3\ \mathfrak{B}_{DS}, \\
k & = block_{\#c}\ \mathfrak{B}_{DS}, \\
n & = block_{\#}\ \mathfrak{B}_{DS}, \\
\\
\langle [\![\vec{C}_1]\!]_{\vec{C}},\ \rho\rangle & = gen\text{-}conds\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\vec{\mathfrak{B}}], \\
\\
[\![\vec{\tau}]\!]_{\vec{\tau}} & = \downarrow_{\vec{\tau}}\ (map\ (\mathbf{K}\ appTp)\ (1\ ..\ (n + \#\ [\![\vec{e}]\!]_{\vec{t}})))
\end{array}$$

$$\frac{}{\begin{array}{l} gen\text{-}try\ \mathrm{try}\ \mathfrak{B}_{DS}\ [\![\vec{C}_2]\!]_{\vec{C}}\ t\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ \mathcal{E}\ = \\ lm_{\varphi}\ [\mathrm{try}]\ \vec{C}_1,\ \vec{C}_2 \texttt{ ===> } \mathrm{try}(\vec{b},\ \vec{e}) = t\ (\mathrm{try}(\vec{\tau})\ :_{fd}\ assign\ \rho\ \mathcal{E}) \end{array}}$$

Since the rest of the conditions may contain non-flat lists the new rule is recursively transformed by lm_{φ}.

Gen-Conds generates the conditions for the first rule for *try*. The variables $\vec{b}$ are the blocks as *try* gets them. The variables $\vec{a}$ correspond to the sublist variables in the original list pattern.

$$\begin{array}{rl} gen\text{-}conds\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\vec{\mathfrak{B}}]\ = & foldr\ gen\text{-}cond\ \langle [\![]\!]_{\vec{C}},\ \Delta_{\rho}\ (\backslash *)\rangle \\ & (zip\ \langle [\![\vec{b}]\!]_{\vec{t}},\ [\![\vec{a}]\!]_{\vec{t}},\ tl\ [\![\vec{b},\ \$nil()]\!]_{\vec{t}},\ map\ (\pi\ 1)\ [\vec{\mathfrak{B}}]\rangle) \end{array}$$

If the block is just a sublist variable

$$gen\text{-}cond\ \langle b,\ a,\ b',\ [\![x^*]\!]_{\vec{t}}\rangle\ \langle [\![\vec{C}]\!]_{\vec{C}},\ \rho\rangle\ =\ \langle [\![\vec{C}]\!]_{\vec{C}},\ x^* \mapsto \$until(b',\ b) \oplus \rho\rangle$$

If the block is a non-empty list of constants followed by a sublists variable

$$\frac{y^* = a\ *,\ \ a' = y^*\ \backslash *}{\begin{array}{r} gen\text{-}cond\ \langle b,\ a,\ b',\ [\![t_1,\ \vec{t},\ x^*]\!]_{\vec{t}}\rangle\ \langle [\![\vec{C}]\!]_{\vec{C}},\ \rho\rangle = \langle [t_1,\ \vec{t},\ y^*] := b : [\![\vec{C}]\!]_{\vec{C}}, \\ x^* \mapsto \$until(b',\ a') \oplus \rho\rangle \end{array}}$$

If the block consists of constants
This can only occurr if the block was the last block of a list of type 7.

$$\frac{type\ t_2 \neq slTp}{gen\text{-}cond\ \langle b,\ a,\ b',\ [\![\vec{t},\ t_2]\!]_{\vec{t}}\rangle\ \langle [\![\vec{C}]\!]_{\vec{C}},\ \rho\rangle = \langle [\vec{t},\ t_2] := b : [\![\vec{C}]\!]_{\vec{C}},\ \rho\rangle}$$

Gen-Try-Fail The default rule for *try*. If the first rule for try was not succesful *next* is called.

$$gen\text{-}try\text{-}fail\ \mathrm{next}\ \mathrm{try}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{e}]\!]_{\vec{t}} = otherwise : [\mathrm{try}]\ \mathrm{try}(\vec{b},\ \vec{e}) = \mathrm{next}(\vec{b},\ \vec{b},\ \vec{e})$$

Pre-Next preprocess the block datastructure for *gen-next*. It sets the number of blocks field in $\mathfrak{B}_{DS}$ to one less when the last element of the last block is not a sublist variable; this happens when the pattern contains a sequence of constants as its tail. This sequence of constants gets assigned a fixed sublist of list. There is no need to reassign it.

$$\begin{array}{ll} \mathit{pre\text{-}next}\ \mathfrak{B}_{DS} = \mathfrak{B}_{DS} & \Leftarrow \mathit{block}_{last}\ (\mathit{last}\ (\pi\ 3\ \mathfrak{B}_{DS})) = \top \\ \mathit{pre\text{-}next}\ \mathfrak{B}_{DS} = \mathit{set}\ 1\ (\mathit{block}_{\#}\ \mathfrak{B}_{DS} - 1)\ \mathfrak{B}_{DS} & \Leftarrow \mathit{block}_{last}\ (\mathit{last}\ (\pi\ 3\ \mathfrak{B}_{DS})) = \bot \end{array}$$

Gen-Next generates a TRS containing the declaration and the rules for the function *next*. For each block *gen-next-i* generates one rule.

$$\frac{\begin{array}{l} \mathfrak{B}_{DS2} = \mathit{pre\text{-}next}\ \mathfrak{B}_{DS}, \\ [\![\vec{\tau}]\!]_{\vec{\tau}} \ = \downarrow [\![]\!]_{\vec{\tau}}\ (\mathit{map}\ (\mathbf{K}\ \mathit{appTp})\ (1\ ..\ (2 \cdot \mathit{block}_{\#}\ \mathfrak{B}_{DS} + \#\ [\![\vec{e}]\!]_{\vec{t}}))) \end{array}}{\begin{array}{l} \mathit{gen\text{-}next}\ \mathrm{next}\ \mathit{try}\ \mathfrak{B}_{DS}\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{b}\,']\!]_{\vec{t}}\ = \\ \mathit{trs}\ [\![\mathrm{next}(\vec{\tau})\ ]\!]_{\vec{fd}}\ (\downarrow_{\vec{\varphi}}\ (\mathit{map}\ (\mathit{gen\text{-}next\text{-}i}\ \mathrm{next}\ \mathit{try}\ \mathfrak{B}_{DS2}\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{b}\,']\!]_{\vec{t}}) \\ \qquad\qquad\qquad (1\ ..\ \mathit{block}_{\#}\ \mathfrak{B}_{DS2}))) \end{array}}$$

Next-i There are three cases. The first and second rules handle the case for the first and last block. We label the cases with the same numbers as in the example for the *set* rule.

Case 1 The first block. If the second block has exactly k (the number of constants in the second to last blocks) elements, all divisions in sublists have been tried. The failure is reported by the constant *$fail*().

$$\frac{k = \mathit{block}_{acc\text{-}consts}\ (\mathit{block}_i\ 2\ \mathfrak{B}_{DS}),\quad c = \mathit{ListN}\ k\ C}{\begin{array}{l} \mathit{gen\text{-}next\text{-}i}\ \mathrm{next}\ \mathit{try}\ \mathfrak{B}_{DS}\ [\![\vec{e}]\!]_{\vec{t}}\ [\![a_1, a_2, \vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{b}\,']\!]_{\vec{t}}\ 1\ = \\ [\mathrm{next}]\ \mathrm{next}(a_1, c, \vec{a}, \vec{b}, \vec{e}) = \mathit{\$fail}() \end{array}}$$

Case 3 If the last block contains more than k elements (the number of constants in the last block) one element can be shifted to the last but first sublist.

$$\frac{\begin{array}{ll} \mathit{block}_{\#}\ \mathfrak{B}_{DS} = n, & \\ k & = \mathit{block}_{acc\text{-}consts}\ (\mathit{block}_i\ n\ \mathfrak{B}_{DS}), \\ c & = \mathit{ListN}^{+}\ (k+1)\ C, \\ [\![\vec{a}_2]\!]_{\vec{t}} & = \mathit{set}\ n\ c\ [\![\vec{a}_1]\!]_{\vec{t}}, \\ [\![\vec{b}_2]\!]_{\vec{t}} & = \mathit{set}\ n\ (\mathit{Tl}\ (\pi\ n\ [\![\vec{b}_1]\!]_{\vec{t}}))\ [\![\vec{b}_1]\!]_{\vec{t}} \end{array}}{\begin{array}{l} \mathit{gen\text{-}next\text{-}i}\ \mathrm{next}\ \mathit{try}\ \mathfrak{B}_{DS}\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}_1]\!]_{\vec{t}}\ [\![\vec{b}_1]\!]_{\vec{t}}\ [\![\vec{b}_1\,']\!]_{\vec{t}}\ n\ = \\ [\mathrm{next}]\ \mathrm{next}(\vec{a}_2, \vec{b}_1, \vec{e}) = \mathit{try}(\vec{b}_2, \vec{e}) \end{array}}$$

Case 2 If the n-th block is neither the first, nor the last then we generate a rule that tests if the n-th block has more than k_1 elements and the $(n+1)$-st block precisely k_2 elements, where k_1 is the accumulated number of constants in the n-th block and k_2 that number for the $(n+1)$-st block.

If that is the case, one element is shifted to the $(n-1)$-st sublist and the sublists after the n-th are reinitialized.

$$
\frac{\begin{array}{ll}
m & = block_{\#}\ \mathfrak{B}_{DS}, \\
n > 1 & = \top, \\
n < m & = \top, \\
\\
[\mathfrak{B}_1, \mathfrak{B}_2, \vec{\mathfrak{B}}] & = drop\ (n-1)\ (\pi\ 3\ \mathfrak{B}_{DS}), \\
\\
k_1 & = block_{acc\text{-}consts}\ \mathfrak{B}_1, \\
k_2 & = block_{acc\text{-}consts}\ \mathfrak{B}_2, \\
\\
c & = ListN^{+}\ (k_1 + 1)\ C, \\
d & = ListN\ k_2\ D, \\
[\![\vec{a}\,']\!]_{\vec{t}} & = set\ n\ c\ (set\ (n+1)\ d\ [\![\vec{a}]\!]_{\vec{t}}), \\
\\
[\![\vec{b}_0]\!]_{\vec{t}} & = take\ (n-1)\ [\![\vec{b}]\!]_{\vec{t}}, \\
\\
[\![b_1, \vec{b}_1]\!]_{\vec{t}} & = drop\ (n-1)\ [\![\vec{b}\,']\!]_{\vec{t}}, \\
C & = b_1 := Tl\ (\pi\ n\ [\![\vec{b}]\!]_{\vec{t}}), \\
[\![\vec{C}]\!]_{\vec{C}} & = \downarrow_{\vec{C}}\ (zip\text{-}with\ (:=)\ [\![\vec{b}_1]\!]_{\vec{t}}\ (zip\text{-}with\ next\text{-}cond\ [\mathfrak{B}_1, \mathfrak{B}_2, \vec{\mathfrak{B}}]\ [\![b_1, \vec{b}_1]\!]_{\vec{t}}))
\end{array}}{\begin{array}{l}
gen\text{-}next\text{-}i\ \mathrm{next}\ try\ \mathfrak{B}_{DS}\ [\![\vec{e}]\!]_{\vec{t}}\ [\![\vec{a}]\!]_{\vec{t}}\ [\![\vec{b}]\!]_{\vec{t}}\ [\![\vec{b}\,']\!]_{\vec{t}}\ n\ = \\
[\mathrm{next}]\ C, \vec{C} ===> \mathrm{next}(\vec{a}\,', \vec{b}, \vec{e}) = try(\vec{b}_0, b_1, \vec{b}_1, \vec{e})
\end{array}}
$$

Next-cond generates the reinitialization of a block. If b is the pointer to the sublist for block $\mathfrak{B}$, then the sublist after b should be k elements less then b, where k is the number of constants in $\mathfrak{B}$.

$$next\text{-}cond\ \mathfrak{B}\ b = (\,Tl\,\hat{}\ k)\ b \Leftarrow k = block_{consts}\ \mathfrak{B}$$

Chapter 13

Transformation FL

Transformation LM, that we defined in the previous chapter, only deals with TRSs in a normalized form. Transformation FL transforms a specification to this normalized (flat) form. All rules of a TRS are translated to rules in which non-deterministic list patterns appear only as the lhs's of assignment conditions. List patterns which have a form that can be directly translated to *$cons*/*$nil* lists (head/tail patterns like $[X, Y, Z*]$) are left as they are.

The import graph for this chapter is shown in Figure 13.1.

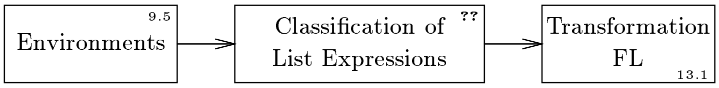


Figure 13.1: Import graph for Chapter 13; Transformation FL

13.1 Transformation FL

Imports Classification-of-List-Expressions$^{(12.3)}$

Definition 13.1.1 (Flat) A *rule* is flat if all its matching terms contain only lists of type 1 or 4. With the exception that the lhs's of assignments may be lists of type 3, 7 or 8 if the rhs is a variable or a list $[x^*]$. A TRS is flat if all its rules are flat. We will denote the language of flat TRSs as $\mathrm{RN2}_{\mathit{fl}}$.

Example The rule $[a]\ a([X*, X]) = b()$ is not flat. A flat version of this rule is $[a]\ [X*, X] := L ===> a(L) = b()$

Exports

Context-free Functions

"fl" "⟦" TRS "⟧" $\rightarrow$ TRS
"fl" "⟦" TERM "⟧" $\rightarrow$ TERM
"fl^{-1}" "⟦" TERM "⟧" $\rightarrow$ TERM

The pre- and postcondition for fl are

$$\mathit{fl}_{pre}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{RN2}_{ll} \qquad \mathit{fl}_{post}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{RN2}_{\mathit{fl}} \qquad \star$$

The following combinators are used to implement the transformation.

Hiddens

Context-free Functions

"$fl_{\mathcal{T}}$" $\to$ Combinator {**type:** TRS $\to$ TRS}
"fl_{φ}" $\to$ Combinator {**type:** RULE $\to$ RULE}
"fl_{C}" $\to$ Combinator {**type:** COND $\to \mathbb{E} \to \mathbb{E}$}
"fl_t^r" $\to$ Combinator {**type:** $\mathbb{E} \to$ TERM $\to \mathbb{E} \times$ TERM}
"fl_t" $\to$ Combinator {**type:** $\mathbb{E} \to$ FUNSYM $\cup$ VAR $\cup$ {list} $\to$ TERM* $\to \mathbb{E} \times$ TERM}
"fl_l" $\to$ Combinator {**type:** $\mathbb{N} \to$ TERM* $\to$ ENV $\to \mathbb{E} \times$ TERM}

Equations

The transformation is carried out by combinator $fl_{\mathcal{T}}$. The signature is not changed by the transformation, so the input and output transformation on terms are identities.

$$fl[\![\mathcal{T}]\!] = \mathcal{T}' \Leftarrow \mathcal{T}' = fl_{\mathcal{T}}\ \mathcal{T} \qquad fl[\![t]\!] = t \qquad fl^{-1}[\![t]\!] = t$$

TRS We start with the top function $fl_{\mathcal{T}}$. It builds a new specification with the old signature and the result of flattening all rules.

$$fl_{\mathcal{T}}\ \mathcal{T} = \mathit{trs}\ (\Sigma\ \mathcal{T})\ (\mathit{map}\ fl_{\varphi}\ (\mathbf{R}\ \mathcal{T}))$$

Rules are flattened by flattening all their matching terms.

$$\begin{aligned} fl_{\varphi}\ \varphi = {} & \mathit{rule}\ (\mathit{kind}\ \varphi)\ (\mathit{tag}\ \varphi)\ (\pi_{\vec{C}}\ \mathcal{E}_2)\ t\ (\mathit{rhs}\ \varphi) \\ & \textbf{when} \\ & \quad \mathcal{E}_1 \qquad = \mathit{foldr}\ fl_C\ (\mathit{new\text{-}env}\ \$fl\ \text{FL})\ (\mathit{conds}\ \varphi), \\ & \quad \langle \mathcal{E}_2, t \rangle = fl_t^r\ \mathcal{E}_1\ (\mathit{lhs}\ \varphi) \end{aligned}$$

Conditions Positive and negative conditions have no matching terms. They are just added to the list of conditions in the environment.

$$fl_C\ C\ \mathcal{E} = C :_C \mathcal{E} \Leftarrow \mathit{new?}\ C = \bot$$

If the condition is an assignment, the rhs is a normal variable and the lhs is a list pattern of type 3, 7 or 8, only the proper subterms of the lhs are flattened.

$$\frac{\mathit{list\text{-}kind}\ [\![\vec{t}]\!]_{\vec{t}} \in [3, 7, 8] = \top,\ \langle \mathcal{E}_2, [\![\vec{t}']\!]_{\vec{t}} \rangle = \mathit{mapAcc}\ fl_t^r\ [\![]\!]_{\vec{t}}\ \mathcal{E}_1\ [\![\vec{t}]\!]_{\vec{t}}}{fl_C\ [\vec{t}] := x\ \mathcal{E}_1 = [\vec{t}'] := x :_C \mathcal{E}_2}$$

A list $[x^*]$ will turn out to be equivalent to a variable in the translation to *\$cons/\$nil* lists. Therefore the case where the rhs of the assignment is $[x^*]$, is treated as if it were a variable.

$$\frac{\mathit{list\text{-}kind}\ [\![\vec{t}]\!]_{\vec{t}} \in [3, 7, 8] = \top,\ \langle \mathcal{E}_2, [\![\vec{t}']\!]_{\vec{t}} \rangle = \mathit{mapAcc}\ fl_t^r\ [\![]\!]_{\vec{t}}\ \mathcal{E}_1\ [\![\vec{t}]\!]_{\vec{t}}}{fl_C\ [\vec{t}] := [x^*]\ \mathcal{E}_1 = [\vec{t}'] := [x^*] :_C \mathcal{E}_2}$$

Otherwise the entire lhs is flattened.

$$\frac{\langle \mathcal{E}_2, t' \rangle = fl_t^r\ \mathcal{E}_1\ t}{fl_C\ t := s\ \mathcal{E}_1 = t' := s :_C \mathcal{E}_2} \quad \textbf{otherwise}$$

Terms fl^r_t flattens all list patterns in a term t and yields the pair $\langle \mathcal{E}, t' \rangle$, where $\mathcal{E}$ is the environment containing the updated list of condititions and t' is the flat version of t.

$$\mathit{fl}^r_t\ \mathcal{E}\ t = \mathit{poAcc}\ \mathit{fl}_t\ [\![]\!]_{\vec{t}}\ \mathcal{E}\ t$$

Variables and applications are flat.

$$\mathit{fl}_t\ \mathcal{E}\ x^{(*)}\ [\![]\!]_{\vec{t}} = \langle \mathcal{E},\ x^{(*)} \rangle \qquad \mathit{fl}_t\ \mathcal{E}\ f\ [\![\vec{t}]\!]_{\vec{t}} = \langle \mathcal{E},\ f(\vec{t}) \rangle$$

List expressions ($[\vec{t}]$) are flattened by fl_l.

$$\mathit{fl}_t\ \mathcal{E}\ \mathit{list}\ [\![\vec{t}]\!]_{\vec{t}} = \mathit{fl}_l\ (\mathit{list\text{-}kind}\ [\![\vec{t}]\!]_{\vec{t}})\ [\![\vec{t}]\!]_{\vec{t}}\ \mathcal{E}$$

Lists of type 1 and 4 can be translated directly to cons/nil patterns; they are not flattened.

$$\mathit{fl}_l\ k\ [\![\vec{t}]\!]_{\vec{t}}\ \mathcal{E} = \langle \mathcal{E},\ [\vec{t}] \rangle \Leftarrow k \in [1,\ 4] = \top$$

For instance, the pattern $[X, b(), Y*]$ is of type 4. It is equivalent to the pattern $\mathit{\$cons}(X, \mathit{\$cons}(b(), L))$.

Lists of type 2, 5 and 6 that is patterns of the form $[\vec{a}, x^*, \vec{b}]$, where $\vec{a}$ does not contain sublist variables and $\vec{b}$ is not empty. These can not be expressed directly as *\$cons*/*\$nil* lists. But they can be split in the list $[\vec{a}, y^*]$ which is of type 4, and the condition $[x^*, \vec{b}] := [y^*]$, where y^* is a new variable. The lhs of this new condition has type 3, 7 or 8 and is therefore flat.

$$\begin{array}{l}
\mathit{fl}_l\ k\ [\![\vec{t}]\!]_{\vec{t}}\ \mathcal{E}_1 = \langle [x^*,\ \vec{b}] := [y^*] :_C \mathcal{E}_2,\ [\vec{a},\ y^*] \rangle \\
\qquad\qquad\qquad \textbf{when} \\
\qquad\qquad\qquad\quad \begin{array}{ll}
k \in [2,\ 5,\ 6] & = \top, \\
\langle [\![\vec{a}]\!]_{\vec{t}},\ [\![x^*,\ \vec{b}]\!]_{\vec{t}} \rangle & = \mathit{split\text{-}while}\ (/\ \mathit{slVar?})\ [\![\vec{t}]\!]_{\vec{t}}, \\
\langle \mathcal{E}_2,\ y \rangle & = \mathit{alloc\text{-}var}\ \mathcal{E}_1, \\
y^* & = y\ *
\end{array}
\end{array}$$

For instance the list $[Y, X*, Z, A*]$ is of type 6. It is replaced by $[Y, C*]$, for some new variable $C*$ and the condition $[X*, Z, A*] := [C*]$ is added as new condition.

Lists of type 3, 7 and 8 have the form $[x^*, \vec{t}]$. These can not be expressed by *\$cons*/*\$nil* lists and need special treatment from transformation *lm*. They are replaced with a new variable x and the condition $[x^*, \vec{t}] := x$ is added.

$$\frac{k \in [3,\ 7,\ 8] = \top,\ \ \langle \mathcal{E}_2,\ x \rangle = \mathit{alloc\text{-}var}\ \mathcal{E}_1}{\mathit{fl}_l\ k\ [\![x^*,\ \vec{t}]\!]_{\vec{t}}\ \mathcal{E}_1 = \langle [x^*,\ \vec{t}] := x :_C \mathcal{E}_2,\ x \rangle}$$

Remark Transformation fl is the identity function on specifications that are in head/tail form. That is, if all list patterns in matching terms are of type 1 or 4 the specification is not changed. This keeps the patterns of matching terms as specific a possible, and thus yields a more efficient reduction machine.

We conclude this module with some example transformations.

Example Rule

$$[\mathit{set}]\ I \stackrel{?}{=} J \texttt{ ===> } \mathit{set}([X*, I, Y*, J, Z*]) = \mathit{set}([X*, I, Y*, Z*])$$

is transformed to

$$[set]\ [X*, I, Y*, J, Z*] := FL\$0, I \stackrel{?}{=} J ===> set(FL\$0) = set([X*, I, Y*, Z*])$$

Rule

$$palindrome([A, A*, A]) = palindrome([A*])$$

is transformed to

$$[A*, A] := [FL\$0*] ===> palindrome([A, FL\$0*]) = palindrome([A*])$$

Chapter 14

Transformation LL

Transformation LL transforms non-left linear TRSs into left-linear TRSs. The import graph for this chapter is shown in Figure 14.1

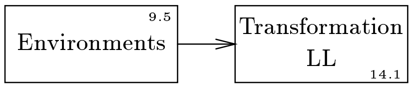


Figure 14.1: Import graph for Chapter 14; Transformation LL

14.1 Transformation LL

Imports Environments$^{(9.5)}$

Transformation ll makes all rules in a TRS left-linear. The definition of left-linearity is

Definition 14.1.1 (Left-Linear) A matching term t is left-linear if each variable occurs at most once in t. A rule φ is left-linear if all its matching terms are left linear. A TRS is left-linear if all its rules are left-linear. We will denote the set of all left-linear TRSs as $\mathrm{RN2}_{ll}$.

Example The rule $[eq]\ A \stackrel{?}{=} B$ ===> $eq(A, B) = true()$ is left-linear. This rule is equivalent to $[eq]\ eq(A, A) = true()$ which is not left-linear, since the lhs has two occurences of A.

The transformation consists of the following functions

Exports

Context-free Functions

"ll" "⟦" TRS "⟧"	$\rightarrow$ TRS
"ll" "⟦" TERM "⟧"	$\rightarrow$ TERM
"ll^{-1}" "⟦" TERM "⟧"	$\rightarrow$ TERM

where the precondition for this tranformation is

$$ll_{pre}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{RN2} \qquad \bigstar$$

and the postcondition is

$$ll_{post}[\![\mathcal{T}]\!] = \mathcal{T} \in \mathrm{RN2}_{ll} \qquad \bigstar$$

The transformation is specified using the following combinators.

Hiddens

Context-free Functions

"$ll_{\mathcal{T}}$" $\to$ Combinator {**type:** TRS $\to$ TRS}
"ll_{φ}" $\to$ Combinator {**type:** RULE $\to$ RULE}
"ll_{C}" $\to$ Combinator {**type:** COND $\to \mathbb{E} \to \mathbb{E}$}
"ll_t^r" $\to$ Combinator {**type:** $\mathbb{E} \to$ TERM $\to \mathbb{E} \times$ TERM}
"ll_t" $\to$ Combinator {**type:** $\mathbb{E} \to$ (FUNSYM $\cup$ VAR $\cup$ {list}) $\to$ TERM* $\to \mathbb{E} \times$ TERM}

Equations

The transformation ll is defined by $ll_{\mathcal{T}}$. The transformation does not change the signature so input terms are not effected.

$$ll[\![\mathcal{T}]\!] = \mathcal{T}' \Leftarrow \mathcal{T}' = ll_{\mathcal{T}}\ \mathcal{T} \qquad ll[\![t]\!] = t \qquad ll^{-1}[\![t]\!] = t$$

TRSs All rules are made left-linear by ll_{φ}. The signature does not change.

$$ll_{\mathcal{T}}\ \mathcal{T} = trs\ (\Sigma\ \mathcal{T})\ (map\ ll_{\varphi}\ (\mathbf{R}\ \mathcal{T}))$$

Rules All matching terms of a rule, i.e. the lhs of the rule and the lhs's of new conditions, are transformed by ll_t^r.

$$\begin{array}{l} ll_{\varphi}\ \varphi = rule\ (kind\ \varphi)\ (tag\ \varphi)\ (\pi_{\vec{C}}\ \mathcal{E}_2)\ t\ (rhs\ \varphi) \\ \qquad \textbf{when} \\ \qquad\quad \mathcal{E}_1 \qquad = foldr\ ll_C\ (new\text{-}env\ \$ll\ \text{LL})\ (conds\ \varphi), \\ \qquad\quad \langle \mathcal{E}_2, t\rangle = ll_t^r\ (set_{vars}\ []\ \mathcal{E}_1)\ (lhs\ \varphi) \end{array}$$

Conditions Positive or negative conditions are not transformed since they have no matching terms.

$$ll_C\ C\ \mathcal{E} = C :_C \mathcal{E} \Leftarrow new?\ C = \bot$$

The lhs of a new condition is transformed by ll_t^r.

$$\frac{\langle \mathcal{E}_2, s'\rangle = ll_t^r\ (set_{vars}\ []\ \mathcal{E}_1)\ s}{ll_C\ s := t\ \mathcal{E}_1 = s' := t :_C \mathcal{E}_2}$$

Terms Given a matching term detect variables occurring more than once, replace them by new variables, and make a new condition identifying the old and new variable.

$$ll_t^r\ \mathcal{E}\ t = poAcc\ ll_t\ [\![]\!]_{\vec{t}}\ \mathcal{E}\ t$$

We are only interested in variables; after the subterms of an application or list have been transformed we reconstruct the application or list.

$$ll_t\ \mathcal{E}\ f\ [\![\vec{t}]\!]_{\vec{t}} = \langle \mathcal{E}, f(\vec{t})\rangle \qquad ll_t\ \mathcal{E}\ list\ [\![\vec{t}]\!]_{\vec{t}} = \langle \mathcal{E}, [\vec{t}]\rangle$$

If a variable did not occur already in the term, the variable is left unchanged and is added to the list of variables in the environment

$$ll_t\ \mathcal{E}\ x^{(*)}\ [\![]\!]_{\vec{t}} = \langle [\![x^{(*)}]\!]_{\vec{t}} +\!\!+_{vars}\ \mathcal{E}, x^{(*)}\rangle \Leftarrow x^{(*)} \in \pi_{vars}\ \mathcal{E} = \bot$$

A variable that occured in the term ($x \in \pi_{\text{vars}}\mathcal{E}1$) is replaced by a new variable (y), and the condition ($x \stackrel{?}{=} y$), that requires the new and old variables to be equal, is added to the environment.

$$\frac{x \in \pi_{vars}\ \mathcal{E}_1 = \top,\ \langle \mathcal{E}_2, y\rangle = alloc\text{-}var\ \mathcal{E}_1}{ll_t\ \mathcal{E}_1\ x\ [\![]\!]_{\vec{t}} = \langle x \stackrel{?}{=} y :_C \mathcal{E}_2, y\rangle}$$

If the variable is a sublist variable the condition should be over lists; sublist variables may not occur as top terms of conditions.

$$\frac{x^* \in \pi_{vars}\ \mathcal{E}_1 = \top,\ \ \langle \mathcal{E}_2, x\rangle = \mathit{alloc\text{-}var}\ \mathcal{E}_1,\ \ y^* = x\ *}{ll_t\ \mathcal{E}_1\ x^*\ [\![]\!]_{\vec{t}} = \langle [x^*] \stackrel{?}{=} [y^*] :_C \mathcal{E}_2, y^* \rangle}$$

Chapter 15

Validation

15.1 From Specification to Compiler Component

Now that we have given a specification for the elemination of associative lists from RN2 specifications, we discuss the compilation of the specifications of the transformations LL, FL and LM to compiler components for the ASFtoC compiler. First, however we define a correctness test for the transformations based on the definition of transformations in Chapter 8.

The following section contains a test suite for the validation of the correctness requirement for transformations as is given in definition 8.3.1. A proof of the correctness would be better, but such a proof would become too large.

15.1.1 Correctness Test

Imports Example-TRSs$^{(9.2)}$ Typecheck-RN2$^{(10.1)}$ Term-Reduction$^{(11.5)}$
Transformation-LL$^{(14.1)}$ Transformation-FL$^{(13.1)}$ Transformation-LM$^{(12.6)}$

We define the composition of the transformations *ll*, *fl* and *lm*. Furthermore, a function *correct?* is defined to test the correctness of the transformation.

Exports

Context-free Functions

"*tr*" "⟦" TRS "⟧"	$\to$ TRS
"*tr*" "⟦" TERM "⟧"	$\to$ TERM
"tr^{-1}" "⟦" TERM "⟧"	$\to$ TERM
"tr_{post}" "⟦" TRS "⟧"	$\to$ Boolean
"*correct?*" "⟦" TRS ";" TERM "⟧"	$\to$ Boolean
test "⟦" NAT "⟧"	$\to$ Term

Variables

"*Nat+*" [*0-9'*]$*$	$\to$ NAT+
[*ABPX*] [*0-9'*]$*$ [$*$]	$\to$ {Term "," }$*$
[*XYZ*]	$\to$ Term

Equations

There is no need to define the precondition; any TRS can be transformed with this transformation. The postcondition for the transformation is ($\in$ CDRN2). The TRS should not contain lists. We check this as follows.

$$\begin{aligned} tr_{post}[\![\mathcal{T}]\!] \;=\; & \forall\ (/\ (\equiv\ lstTp))\ (\mathit{flatMap}\ args\ (\Sigma\ \mathcal{T})) \\ & \wedge \\ & \forall\ (/\ list?)\ (\mathit{flatMap}\ subterms\ (\mathit{flatMap}\ terms\ (\mathbf{R}\ \mathcal{T}))) \end{aligned}$$

We then define the transformation *tr* as the composition of the transformations *ll*, *fl* and *lm*.

$$\begin{aligned} tr[\![\mathcal{T}]\!] &= lm[\![fl[\![ll[\![\mathcal{T}]\!]]\!]]\!] \\ tr[\![t]\!] &= lm[\![fl[\![ll[\![t]\!]]\!]]\!] \\ tr^{-1}[\![t]\!] &= ll^{-1}[\![fl^{-1}[\![lm^{-1}[\![t]\!]]\!]]\!] \end{aligned}$$

Correct? The predicate *correct?* checks wether the transformation works correctly for a given TRS and input TERM.

$$\frac{\mathcal{T}' = tr[\![\mathcal{T}]\!]}{correct?[\![\mathcal{T};\, t]\!] = tc[\![\mathcal{T}]\!] \rightarrow (tr_{post}[\![\mathcal{T}']\!] \wedge (tc[\![\mathcal{T},\, t]\!] \rightarrow (tr^{-1}[\![\mathcal{T}'[\![tr[\![t]\!]]\!]]\!] \equiv \mathcal{T}[\![t]\!])))}$$

With this correctness test and the examples in Section 9.2 we now specify some tests that should (and do) all reduce to $\top$.

$$\begin{aligned} test[\![2]\!] &= correct?[\![example[\![0,\,2]\!];\ plus(succ(succ(zero())),\ succ(zero()))]\!] \\ test[\![5]\!] &= correct?[\![example[\![0,\,5]\!];\ sum([zero(),\ succ(zero()),\ succ(zero())])]\!] \\ test[\![6]\!] &= correct?[\![example[\![0,\,6]\!];\ revers([a(),\ b(),\ c(),\ d(),\ e()])]\!] \\ test[\![7]\!] &= correct?[\![example[\![0,\,7]\!];\ palindrome([a(),\ b(),\ c(),\ b(),\ a()])]\!] \\ test[\![81]\!] &= correct?[\![example[\![0,\,8]\!];\ set([a(),\ b(),\ a(),\ b(),\ a(),\ c(),\ a()])]\!] \end{aligned}$$

$$\begin{aligned} test[\![82]\!] \;=\; correct?[\![example[\![0,\,8]\!];\ set([&set([a(),\ b(),\ a()]), \\ &set([a(),\ a(),\ b(),\ a()]), \\ &set([a(),\ b(),\ a(),\ b()])])]\!] \end{aligned}$$

15.1.2 A New Compiler Component

Using the existing ASFtoC compiler, the specification presented here is compiled to a C program. This C program can be integrated in the compiler as a new component that transforms RN2 specifications to list-less specifications.

15.2 Validation of the Requirements

We check if the requirements on transformations, given in Chapter 8, are satisfied.

Requirement 1 (Size of the TRS) Transformed TRSs are bigger than the original TRSs. We do not have any measurements (see point 4 below). However, we can remark the following things:

- All transformed specifications contain the TRS $\mathcal{T}_{cons}$ (Section 12.1).
- For a list pattern like $[X*, X]$ the function *$last1* (one declaration and two rules) are added to the TRS. Only one function is added for all patterns with a constant tail of n terms. Since an average specification will not use many different patterns like this, these functions are negligible.
- For each list pattern, with $n \geq 2$ sublist variables, two function declarations and $2 + n$ rewrite rules are added to the TRS.

Requirement 2 (Reduction Time) Test with transformed TRSs have shown that the reduction time decreases with a factor 2.5 for reductions involving non-deterministic list patterns.

Requirement 3 (Memory Usage) The same tests showed that no extra memory is used by the simulation of list matching in the transformed TRSs.

Requirement 4 (Complexity of the Specification) The specifications for the three transformations have been compiled and tested, from the meta-environment, on example RN2 specifications. Integration in the compiler is not yet done.

Chapter 16

Concluding Remarks

We have given an algebraic specification of syntax and semantics of the term rewrite system specification formalism RN2. Subsequently we defined three transformations on RN2 that eliminate associative lists from TRSs. In the definition of the transformations we used the combinatory style of specification developed in Part I. The specification was presented in a literate programming style.

16.1 The Transformations

We defined the syntax of RN2 by a hierarchy of small extensions to Basic-RN2. The syntax is embedded in the combinatory language of Part I. Several basic combinators operating on RN2 constructs were defined. Then the static constraints on RN2 specifications and the dynamic semantics of RN2 (reduction of a term by a TRS) were specified.

The specification of the semantics serves two purposes. The first is to get a clear understanding of RN2. The second is to have a tool to test the correctness of transformations. The definition of transformations includes a correctness constraint. This constraint is used as a test to validate transformations. Furthermore, efficiency constraints for transformations were given.

Then we defined three syntactic transformations on TRS specifications. Transformation LL removes non-left-linearity from TRSs, transformation FL flattens nested occurrences of lists and transformation LM eliminates associative lists from TRSs by representing lists by the constructors *$cons* and *$nil*.

Tests have shown that executables resulting from encoding list matching in the TRS are a factor 2.5 slower than executables with hardwired list matching. The optimizations planned for ARM programs will probably reduce this factor.

The specifications for the three transformations have been compiled to executables and can be integrated in the existing compiler as new compiler components.

16.2 Use of Combinatory Algebraic Specification

The specification makes use of the combinatory algebraic specification techniques developed in Part I. The effort it took to develop the language and library was compensated by the flexibility they provided in specifying the transformations.

Similar functionality could have been developed without the combinatory style, but it would have been specific for the use in transformations.

The entire specification is quite large. However, the largest part of it is reusable. The tools developed in Part I can be reused in any specification. The basic combinators for RN2 that were defined in Chapter 9 can be reused in the specification of other transformations.

The application of the combinatory techniques in the specification of transformations helped to develop the library of list and tree operations. For instance, the tree operations in Chapter 6 were designed to generalize recursive operations on RN2 terms. The generic definition makes these operations applicable to any other sort.

Disadvantages The specification could behave more efficient as a TRS, as we showed in Chapter 7, if first-order techniques were used.

In Chapter 9 we defined many combinators for the construction of RN2 constructs. The syntax and equations for these combinators could be derived automatically from the syntax. Either by generating syntax rules and equations [Kli92] or by a generic mechanism [Gan93].

The entire tools library is imported. Therefore, more combinators are imported in the specification of transformations than are actually used. Selection of just those combinators that are actually used would imply a drastic redesign of the library and a more complex import graph. Furthermore, this would result in less flexibility in specifying.

This problem can be resolved by defining an operation on specifications that computes the set of functions used in the rules for a particular function. This operation can be applied to the RN1 specifications derived from a specification to extract only those rules that are actually needed. For instance, in the specification of a transformation, we are only interested in the syntax of RN2 and the transformation function.

Such an operation is much like the export operator of Module Algebra [BHK90]. All functions not in the exported signature are hidden. Hidden functions that are not used in the definition of functions in the exported signature can be removed from the specification. (This operator is different from the exports section of an ASF+SDF module; the export operator is applied 'from outside' a module and hides also functions that were imported into the module.)

Higher-Order Matching Despite the use of higher-order functions, the specification is still very algorithmic. Explicit traversal of constructs is done in the transformations. A more declarative style of specification can be reached with the higher-order matching techniques studied in [Hee92b].

As an illustration, consider the following equation, that makes a non-deterministic list pattern inside a term on the lhs of an assignment condition into a top term of a new assignment condition.

$$\frac{x \text{ is a fresh variable}}{\begin{array}{l}[f]\ \vec{C}_1, t([\vec{a}, x^*, \vec{b}, y^*, \vec{c}]) := s, \vec{C}_2 \\ \quad \Rightarrow a = b\end{array} = \begin{array}{l}[f]\ \vec{C}_1, t(x) := s, [\vec{a}, x^*, \vec{b}, y^*, \vec{c}] := x, \vec{C}_2 \\ \quad \Rightarrow a = b\end{array}}$$

Here, $t(\bullet)$ is a second order variable that matches with a term and $\bullet$ matches a subterm of that term.

Bibliography

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition (1984) edition, 1984.

[Bet91] I. Bethke. Lambda-calculus and combinatory logic. Department of mathematics and computer science, University of Amsterdam, 1991. After notes by A.S. Troelstra. Course notes for the course 'lambda-calculus' taught by A.S. Troelstra, 1991–1992.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BHK90] J.A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1958. With two sections by William Craig.

[CHS72] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume 2 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1972.

[Deu92a] A. van Deursen. Specification and generation of a λ-calculus environment. Report CS-R9233, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992. Also appeared in J.L.G. Dietz (ed), Proceedings CSN'92, pages 14-26.

[Deu92b] Arie van Deursen. A numerical package for ASF+SDF. Unpublished document, 1992.

[Gan93] Job Ganzevoort. A generic term representation. GIPE technical meeting, January 1993.

[Hee92a] J. Heering. Implementing higher-order algebraic specifications. In *Proceedings of the 1992 Workshop on the lambdaProlog Programming Language,Report MS-CIS-92-86*, pages 141–157. University of Pennsylvania, Philadelphia, 1992. Published as report MS-CIS-92-86.

[Hee92b] J. Heering. Second-order algebraic specification of static semantics. Technical Report CS-R9254, CWI, Amsterdam, December 1992.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

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

[HK89] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 283–297. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 6.

[Kli90] P. Klint. A meta-environment for generating programming environments. Technical Report CS-R9064, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

[Kli92] P. Klint. Higher-order functional programming in ASF+SDF? (An ongoing discussion with Jan Heering). GIPE technical meeting., 1992.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984. Reprinted as Chapter 4 of [Knu92].

[Knu92] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, 1992.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Also appeared as Report P9202, University of Amsterdam.

[KW93] J.F.Th. Kamperman and H.R. Walters. ARM. Abstract Rewriting Machine. Technical Report CS-9330, CWI, Amsterdam, 1993.

[Mei91] K. Meinke. Equational specification of abstract types and combinators. Technical Report CSR 11-91, University of Wales. University College of Swansea. Computer Science Division, September 1991. To appear in Proc. Computer Science Logic 1991.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1987.

[PL93] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a tutorial*. Prentice-Hall, 1993.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[SW85] Daniel F. Stubbs and Neil W. Webre. *Data Structures. With Abstract Data Types and Pascal*. Brooks/Cole, Monterey, California, 1985.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[Wie91] F. Wiedijk. *Persistence in Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[WK93] H.R. Walters and J.F.Th. Kamperman. The self-fulfilling prophecy. Design and implementation of a compiler for algebraic specifications. Technical report, CWI, 1993. Draft.