

The Definition of RML

Version 2

Mikael Pettersson
Department of Computer and Information Science
Linköping University, Sweden
and
INRIA Sophia Antipolis, France

1998-04-13

Contents

Preface to Version 2

This second version of the definition of Relational ML (RML) incorporates a number of changes to Version 1 [?, Appendix A]. These changes are summarised below.

Summary of changes

Minor syntactic changes: colon (`:`) is used in place of the `interface` keyword; the user's main module is now `Main` instead of `main`; the module containing standard bindings is now `RML` instead of `rml`.

The `default` keyword has been added to allow users to specify default rules to apply if all the previous rules in a relation fail. Although this makes no difference either to the static or the dynamic semantics, it is important from a stylistic point of view to 'declare' one's intentions rather than silently relying on the determinate proof search procedure.

The `fail` procedure has been replaced by the `fail` keyword, which is placed syntactically in the result part of the conclusion of an inference rule. This eliminates the need to introduce dummy return values for the benefit of the type checker.

Logical variables are now separate objects with explicit types. The dynamic semantics has been simplified by eliminating all implicit dereferencing operations. The type system does *not* use SML-style imperative types since it already subsumes Wright's [?] approach.

Equality types à la SML have been added, entailing significant changes to the static semantics.

The limited form of polymorphic recursion for relations, available when their declarations had explicit types, has been removed.

The old `var = exp` goal has been split into two separate constructs. The new form `let pat = exp` is used for local bindings, while the original form continues to express an equality constraint. Local variables in rules are now allowed to shadow module-level variables.

Miscellaneous changes to the set of predefined types and operations: built-in indexing relations `list_nth` etc. are now 0-based; `print` now only accepts strings instead of arbitrary values.

Type and value declarations may now be written in any order in module interfaces and bodies. A dependency analysis is performed to recover a suitable sequential ordering of the declarations. After this reordering, the standard ML-style static elaboration phase is applied.

Sophia Antipolis, April 1998

1 Introduction

This document formally defines RML – the Relational Meta-Language – using Natural Semantics as the specification formalism.

RML is intended as an executable specification language for experimenting with and implementing Natural Semantics. The rationale for the design of the language, and hints on how it may be implemented, are not included here, but may be found in the author’s thesis [?].

The style of this document was greatly influenced by the formal definition of the Standard ML language and notation used in denotational semantics. See [?, ?, ?] for further examples on the kind of Natural Semantics used here.

1.1 Differences to SML

RML is heavily influenced by Standard ML, both in the language itself and in its definition. Below we summarize some of the technical differences between these languages.

RML’s relations are *n*-to-*m*-ary, not 1-to-1 as functions in SML are. Also, RML’s **datatype** constructors are *n*-ary rather than just unary.

The **withtype** construct can introduce a number of type aliases together with a **datatype** declaration. RML expands these *sequentially* instead of simultaneously, which allows limited dependencies between aliases.

The RML module system is much simpler than that in SML. A module is an environment of type and value bindings. At the top level of a module, no type identifier, data constructor, or variable may be multiply bound, and in a program, no module identifier may be multiply bound. For stand-alone applications, RML defines the entry point to be module **Main**’s relation **main**, which must be of type **string list => ()**.

Both SML and RML introduce a unique tag to represent each user-level **datatype** in the type system. Such a tag is known as a *type name*, but is *not* the type identifier used in the **datatype** declaration. In RML, type names are (essentially) pairs (*module id*, *type id*). Due to its simpler module system, these pairs are guaranteed to be unique.

In several cases, the definition of RML uses explicit inference rules to define features, where the definition of SML relies on comments in the accompanying text.

Like Haskell, but unlike SML, RML allows declarations to be written in any order. A reordering phase is used to recover, when possible, the corresponding SML-style program with definitions before uses, and explicitly marked groups of mutually dependent declarations.

2 Notation for Natural Semantics

2.1 Lexical definitions

Lexical definitions are made primarily using regular expressions. These are written using the following notation¹, in which alternatives are listed in decreasing order of precedence:

c	denotes the character c , if c is not one of $. * + ? () \{ \} \backslash " [] ^$
$\backslash \mathbf{t}$	denotes a tab character (ASCII 9)
$\backslash \mathbf{n}$	denotes a newline character (ASCII 10)
$\backslash ddd$	denotes the single character with number ddd , where ddd is a 3-digit decimal integer in the interval $[0, 255]$
$\backslash c$	denotes the character c
$[\dots]$	denotes the set of characters listed in \dots
$[^ \dots]$	denotes the complement of $[\dots]$
$c_1 - c_2$	(within $[\dots]$) denotes the range of characters from c_1 to c_2
$.$	denotes any character except newline
$"x"$	denotes the string of characters x
$\{x\}$	equals the expression bound to the identifier x
(x)	equals x
x^*	denotes the Kleene closure of x
x^+	denotes the positive closure of x
$x?$	denotes an optional occurrence of x
$x\{n\}$	denotes n repetitions of x , where n is a small integer
$x\{n_1, n_2\}$	denotes between n_1 and n_2 repetitions of x
xy	denotes the concatenation of x and y
$x y$	denotes the disjunction of x and y

2.2 Syntax definitions

Syntax definitions are made using extended context-free grammars. The following conventions apply:

- The brackets $\langle \rangle$ enclose optional phrases.
- Alternative forms for each phrase class are listed in *decreasing* order of precedence.
- ε denotes an empty phrase.
- \dots denotes repetition. It is never a literal token.
- Constraints on the applicability of a production may be added.

¹This notation coincides with the one used by the `ml-lex` scanner generator.

- A production may be indicated as being left (right) (non) associative by adding the letter L (R) (N) to its right.
- References to literal tokens are printed in **this** style.
- References to syntactic phrases or non-literal lexical items are printed in *this* style.

2.3 Sets

If A is a set, then $\text{Fin } A$ denotes the set of finite subsets of A .

2.4 Tuples

Tuples are ordered heterogeneous collections of fixed finite length.

(x_1, \dots, x_n) a tuple t formed of x_1 to x_n , in that order
 k of t projection; equals x_k if $t = (x_1, \dots, x_k, \dots, x_n)$
 $T_1 \times \dots \times T_n$ the type $\{(x_1, \dots, x_n) ; x_i \in T_i \ (1 \leq i \leq n)\}$

The notation k of t is sometimes extended to x of t , where x is a meta-variable ranging over T_k , and t is of type $T_1 \times \dots \times T_k \times \dots \times T_n$. There must be only one occurrence of the type T_k in T_1, \dots, T_n .

2.5 Finite Sequences

Sequences are ordered homogeneous collections of varying, but always finite, length.

\square	the empty sequence
$x :: s$	the sequence formed by prepending x to sequence s
$s @ s'$	the concatenation of sequences s and s'
$x^{(n)}$ or $[x_1, \dots, x_n]$	the sequence $x_1 :: \dots :: x_n :: \square$
x^*	a sequence of 0 or more x 's
x^+	a sequence of 1 or more x 's
$s \downarrow k$	the k 'th element of sequence s (1-based)
$s \setminus k$	the sequence s with its k 'th element removed
$\#s$	the length of sequence s
$\rightleftharpoons s$	the reversal of sequence s
$x \in s$	membership test, $\exists k \in [1, \#s] : s \downarrow k = x$
$s \subseteq s'$	$\forall k \in [1, \#s] : s \downarrow k \in s'$
T^k	the type of all T -sequences of length k
T^* or $\cup_{k \geq 0} T^k$	the type of all finite T -sequences
T^+ or $\cup_{k \geq 1} T^k$	the type of all non-empty finite T -sequences

2.6 Finite Maps

If A and B are sets, then $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B . The domain and range of a

finite map, f , are denoted $\text{Dom } f$ and $\text{Ran } f$. A finite map can be written explicitly in the form $\{a_1 \mapsto b_1, \dots, a_k \mapsto b_k\}, k \geq 0$. The form $\{a \mapsto b ; \phi\}$ stands for a finite map f whose domain is the set of values a which satisfy the condition ϕ , and whose value on this domain is given by $f(a) = b$. If f and g are finite maps, then $f + g$ (f *modified by* g) is the finite map with domain $\text{Dom } f \cup \text{Dom } g$ and values

$$(f + g)(a) = \text{if } a \in \text{Dom } g \text{ then } g(a) \text{ else } f(a).$$

2.7 Substitutions

If t is a term with variables V of type T , and f is a finite map of type $V \xrightarrow{\text{fin}} T$, then f can be used as a *substitution* on t . The expression tf denotes the effect of substituting free variables v_i in t by $f(v_i)$, when $v_i \in \text{Dom } f$. The definition of ‘free’ variables depends on the type of the term t . Substitutions are extended element-wise to finite sequences.

2.8 Disjoint Unions

The type $T_1 \cup \dots \cup T_n$ denotes the *disjoint* union of the types T_1, \dots, T_n . Let x be a meta-variable ranging over a disjoint union type T , and x_i range over its summands T_i .

An x_i is injected into T by the expression x_i in T .

Membership test and projection are normally expressed using pattern-matching syntax. Using a meta-variable x_i in a binding position in a function or relation, where the binding position is of type T , constrains an argument to be in the summand T_i ; moreover, the formal parameter x_i is bound to the projected value in T_i .

2.9 Relations

A relation is a (in general infinite) set of tuples, i.e. a subset of some product $T_1 \times \dots \times T_n$. It is characterized by a *signature* and is defined by a finite set of *inference rules*.

2.9.1 Signatures

A signature is used to declare the form and type of a relation. It is written as a non-empty sequence of meta-variables, with some auxiliary symbols inserted between some of them. Let x_1, \dots, x_n be meta-variables for the types T_1, \dots, T_n . Then a signature whose sequence of meta-variables is $x^{(n)}$, declares a relation over $T_1 \times \dots \times T_n$.

When a relation is seen as defining logical propositions, as is typical for natural semantics, signatures are usually called *judgements*.

Occasionally, the place of a meta-variable will be replaced by a ‘proto-type’ pattern of the form e/e' , which denotes an anonymous type $T \cup T'$, where T (T') is the type of e (e').

The auxiliary symbols inserted in a signature have no semantic effect, other than to make the signature easier to read, and to disambiguate different relations having the same type.

Example: Let ME , TE , θ , and $longtycon$ be the meta-variables for the types $ModEnv$, $TyEnv$, $TypeFcn$, and $longTyCon$ respectively. Then $ME, TE \vdash longtycon \Rightarrow \theta$ is a signature for a relation over $ModEnv \times TyEnv \times longTyCon \times TypeFcn$.

2.9.2 Instances

An instance of a signature is formed by instantiating the meta-variables with expressions (or patterns) of appropriate types.

Groups of relations are often viewed as defining a special-purpose logic. In this case, instances are referred to as *propositions* or *sequents*.

2.9.3 Inference Rules

The contents (set of tuples) of a relation is specified using a finite set of inference rules of the form:

$$\frac{\text{premises}}{\text{conclusion}} \quad (\text{label})$$

The conclusion is written as an instance of the signature of the relation. The premises impose additional conditions, typically by checking that certain values occur in other relations, that certain values are equal (or not equal), or that certain values occur (or do not occur) in some set or sequence. When the premises are true, the conclusion (the existence of an element in the relation) can be inferred.

We additionally require relations to be *determinate*: for every element in the relation, *exactly one* of the relation’s inference rules must hold.

We sometimes use $_$ in the place of a meta-variable. This syntax is used to make it clear that a particular value is ignored.

Inference rules are often labelled, as indicated by the label written to the right.

In the rules, phrases bracketed by $\langle \rangle$ are *optional*. In an instance of a rule, either all or none of the options must be present. This convention, motivated by optional phrases in syntax definitions, allows a reduction in the number of rules.

2.10 Example

These declarations are given for natural and binary numbers:

$n \in \text{Nat}$	natural numbers (primitive)
$b \in \text{Bin} ::= 0 \mid 1 \mid b0 \mid b1$	binary numbers

Here is the specification for a relation expressing a mapping from binary to natural numbers. By convention, we write the relation's signature in a box above and to the right of the rules. We also indicate the type of the main object inspected by writing its name above and to the left of the rules.

Binary Numbers

$b \Rightarrow n$

$$\frac{}{0 \Rightarrow 0} \quad (1)$$

$$\frac{}{1 \Rightarrow 1} \quad (2)$$

$$\frac{b \Rightarrow n}{b0 \Rightarrow 2n} \quad (3)$$

$$\frac{b \Rightarrow n}{b1 \Rightarrow 2n + 1} \quad (4)$$

Relations are often used in a directed manner. For example, a query $b \Rightarrow n$ is typically used when b is known to *compute* n .

3 Lexical Structure

This section defines the lexical structure of RML.

3.1 Reserved Words

The following are the reserved words, including special symbols. The word `--` represents all words generated by the regular expression `-(-)+`.

```
and as axiom datatype default end eqtype fail let
module not of relation rule type val with withtype
& ( ) * , -- . : :: = => [ ] _ |
```

3.2 Whitespace and Comments

The blank, tab, linefeed, carriage return, and formfeed characters are treated as whitespace characters. Whitespace characters between tokens serve as separators, but are otherwise ignored.

```
white = [\ \t\n\013\012]
```

A comment is any character sequence within comment brackets `(*)` in which comment brackets are properly nested. It is an error for a comment bracket to be unmatched. A comment is equivalent to a single blank character.

```
comment ::= ( * skipto * )
skipto * ) ::= * after *
                | ( after (
                | [ ^ * ( ] skipto * )
after * ::= )
                | * after *
                | ( after (
                | [ ^ ) * ( ] skipto * )
after ( ::= * skipto * ) skipto * )
                | ( after (
                | [ ^ * ( ] skipto * )
```

3.3 Integer constants

The token class `ICon`, ranged over by *icon*, denotes integer constants. An integer constant may be written in decimal or hexadecimal base, optionally preceded by a negation sign (`-` or `~`). The precision of integers is implementation-dependent. Examples: `34 0x22 -1`.

```
ddigit = [0-9]
decint = [-~]? {ddigit} +
xdigit = [0-9a-fA-F]
hexint = [-~]? "0x" {xdigit} +
icon = {decint} | {hexint}
```

3.4 Real constants

The token class RCon, ranged over by *rcon*, denotes real constants. A real constant is written as a decimal integer constant, followed by a fraction and an exponent. Either the fraction or the exponent, but not both, may be omitted. The precision of reals is implementation-dependent. Examples:

```
0.7  3.25E5  3E-7  .
fraction = ". "{ddigit}+
exponent = "[eE]{decint}"
rcon    = "{decint"}({fraction}{exponent}?|{exponent})
```

3.5 Character constants

The token class CCon, ranged over by *ccon*, denotes character constants. An underlying alphabet of 256 distinct characters numbered 0 to 255 is assumed, such that the characters numbered 0 to 127 coincide with the ASCII character set. A character constant is written as `#"`, followed by a character descriptor, and terminated by `"`. A character descriptor is either a single printing character, or `\` followed by one of these escape sequences:

`\ddd` A sequence of three decimal digits, denoting a character number in the interval `[0, 255]`.

`\^c` The control character *c*, where *c* may be any character with number 63–95. The number of `\^c` is 64 less than the number of *c*, modulo 128, mapping `\^?` to 127 (delete), and `\^A–\^_` to 0–31.

`\c` A single escape character *c*, with the following interpretations:

```
\\  =  92  (backslash)
\"  =  34  (double quote)
\n  =  10  (linefeed)
\r  =  13  (carriage return)
\t  =   9  (tab)
\f  =  12  (formfeed)
\a  =   7  (alert)
\b  =   8  (backspace)
\v  =  11  (vertical tab)
```

```
echar = ["\\nrtfabv]"
cntrl = ["?–_]"
escseq = "{ddigit}{3}| \"^\"{cntrl}|{echar}"
pchar = ["\ -!#-[^~\128–\255]| \""]
cdesc = "{pchar}|\\{escseq}"
ccon = "#\"\"{cdesc}\""
```

Examples of character constants: `#"n"` `#"\n"` `#"\010"` `#"^J"` .

3.6 String constants

The token class SCon, ranged over by *scon*, denotes string constants. A string constant is written as a sequence of string items enclosed by a pair of double quotes ". A string item is either a character descriptor, denoting a single character, or a *gap*, denoting an empty sequence of characters. Examples: "thirty-four is 3\ 4" "TAB is \t" .

$$\begin{aligned} gap &= \\{white}+\\ \\ sitem &= \{cdesc\}|\{gap\} \\ scon &= \\{"sitem"}*\\ \end{aligned}$$

3.7 Identifiers

The token class Id, ranged over by *id*, denotes identifiers. An identifier is written as a non-empty sequence of ASCII letters, digits, primes, or underscores, starting with a letter. An instance of the regular expression for *id* that coincides with a reserved word is interpreted as that reserved word, not as an identifier. Examples: cons g4711' .

$$\begin{aligned} alpha &= [A-Za-z] \\ alnum &= \{alpha\}|[_'0-9] \\ id &= \{alpha\}\{alnum\}* \end{aligned}$$

3.8 Type Variables

The token class TyVar, ranged over by *tyvar*, denotes type variables. A type variable is written as an identifier, prefixed by one or more primes. The subclass EtyVar of TyVar, the *equality* type variables, consists of those which start with two or more primes. Examples: 'a ''key' .

$$tyvar = '''+\{alpha\}\{alnum\}*$$

3.9 Lexical analysis

Lexical analysis maps a program to a sequence of items, in left to right order. Each item is either a reserved word, an integer, real, character, or string constant, an identifier, or a type variable. Whitespace and comments separate items but are otherwise ignored – except within character and string constants. At each stage, the longest recognizable item is taken.

4 Syntactic Structure

This section defines the syntax of RML.

4.1 Derived Forms, Full and Core Grammar

First the *full* (concrete) syntax is defined in figures ?? to ??. In these rules, standard *derived* forms are marked with (*). Such forms are subject to term-rewriting transformations specified in figures ?? to ??. The resulting *core* syntax is defined in figures ?? to ??. Figure ?? defines the implicit syntax of programs, i.e. sequences of modules.

The derived forms for **with** specifications and declarations use string literals to indicate the names of the files in which the intended external modules are located. In the core syntax, the interface of such an external module is made explicit (figure ??).

4.2 Ambiguity

The full grammar as given is highly ambiguous: the reasons are that an identifier may stand either for a (short) constructor or a variable binding in a pattern, and in an expression it may stand for a constructor or a variable reference. Environment information is in general necessary to determine the meaning of an identifier in these contexts.

A parser could produce an ambiguous syntax tree where these possibilities have been joined to a single ‘unknown identifier’ case. A type checker could then construct an unambiguous syntax tree using the type information it has access to.

The static semantics rules given later assume an unambiguous core syntax tree, but also verify that the identifier classification was correct.

$tycon \in \text{TyCon}$	$::= id$	type constructor
$con \in \text{Con}$	$::= id$	value constructor
$var \in \text{Var}$	$::= id$	value variable
$modid \in \text{ModId}$	$::= id$	module name
$longtycon \in \text{longTyCon}$	$::= \langle modid . \rangle tycon$	long type constructor
$longcon \in \text{longCon}$	$::= \langle modid . \rangle con$	long value constructor
$longvar \in \text{longVar}$	$::= \langle modid . \rangle var$	long value variable
$lit \in \text{Lit}$	$::= ccon$	character constant
	$ icon$	integer constant
	$ rcon$	real constant
	$ scon$	string constant

Figure 1: Full grammar: Auxiliaries

$tyvarseq \in \text{TyVarSeq}$	$::= \varepsilon$	empty (*)
	$tyvar$	singleton (*)
	$(tyvar_1, \dots, tyvar_n)$	$n \geq 1$
$ty \in \text{Ty}$	$::= tyvar$	variable
	$\langle tyseq \rangle longtycon$	construction (*)
	$ty_1 * \dots * ty_n$	tuple, $n \geq 2$
	$tyseq_1 \Rightarrow tyseq_2$	relation
	(ty)	(*)
$tyseq \in \text{TySeq}$	$::= ()$	empty
	ty	singleton (*)
	(ty_1, \dots, ty_n)	sequence, $n \geq 1$

Figure 2: Full grammar: Types

$pat \in \text{Pat}$	$::= _$	wildcard
	lit	literal
	$[pat_1, \dots, pat_n]$	list, $n \geq 0$ (*)
	$longcon$	constant
	var	variable (*)
	$longcon patseq$	structure
	(pat_1, \dots, pat_n)	tuple, $n \neq 1$
	$pat_1 :: pat_2$	cons, R (*)
	$var \text{ as } pat$	binding
	(pat)	(*)
$patseq \in \text{PatSeq}$	$::= ()$	empty
	pat	singleton (*)
	(pat_1, \dots, pat_n)	sequence, $n \geq 1$

Figure 3: Full grammar: Patterns

$exp \in \text{Exp}$	$::= lit$	literal
	$[exp_1, \dots, exp_n]$	list, $n \geq 0$ (*)
	$longcon$	constant
	$longvar$	variable
	$longcon expseq$	structure
	(exp_1, \dots, exp_n)	tuple, $n \neq 1$
	$exp_1 :: exp_2$	cons, R (*)
	(exp)	(*)
$expseq \in \text{ExpSeq}$	$::= ()$	empty
	exp	singleton (*)
	(exp_1, \dots, exp_n)	sequence, $n \geq 1$

Figure 4: Full grammar: Expressions

$goal \in \text{Goal}$	$::= longvar \langle expseq \rangle \langle => \langle patseq \rangle \rangle$	call (*)
	$var = exp$	equality
	$let\ pat = exp$	binding
	$not\ goal$	negation
	$goal_1 \ \& \ goal_2$	sequence
	$(goal)$	(*)
$result \in \text{Result}$	$::= \langle expseq \rangle$	return (*)
	$fail$	fail
$clause \in \text{Clause}$	$::= rule \langle goal \rangle -- var \langle patseq \rangle \langle => result \rangle$	rule (*)
	$axiom\ var \langle patseq \rangle \langle => result \rangle$	axiom (*)
	$clause_1\ clause_2$	sequence
$rule \in \text{Rule}$	$::= clause \langle default\ clause \rangle$	rules (*)

Figure 5: Full grammar: Goals, Clauses, and Rules

$conbind \in \text{ConBind}$	$::= con \langle of\ ty_1 * \dots * ty_n \rangle$	$\langle n \geq 1 \rangle$
	$conbind_1 \mid conbind_2$	
$datbind \in \text{DatBind}$	$::= tyvarseq\ tycon = conbind$	
	$datbind_1 \ \mathbf{and} \ datbind_2$	
$typbind \in \text{TypBind}$	$::= tyvarseq\ tycon = ty$	
	$typbind_1 \ \mathbf{and} \ typbind_2$	
$withbind \in \text{WithBind}$	$::= \langle \mathbf{withtype}\ typbind \rangle$	
$relbind \in \text{RelBind}$	$::= var \langle : ty \rangle = rule$	(*)
	$relbind_1 \ \mathbf{and} \ relbind_2$	
$spec \in \text{Spec}$	$::= \mathbf{with}\ scon$	(*)
	$\mathbf{type}\ tyvarseq\ tycon$	
	$\mathbf{eqtype}\ tyvarseq\ tycon$	
	$\mathbf{type}\ typbind$	
	$\mathbf{datatype}\ datbind\ withbind$	
	$\mathbf{val}\ var : ty$	
	$\mathbf{relation}\ var : ty$	
	$spec_1\ spec_2$	
$dec \in \text{Dec}$	$::= \mathbf{with}\ scon$	(*)
	$\mathbf{type}\ typbind$	
	$\mathbf{datatype}\ datbind\ withbind$	
	$\mathbf{val}\ var = exp$	
	$\mathbf{relation}\ relbind$	
	$dec_1\ dec_2$	
$interface \in \text{Interface}$	$::= \mathbf{module}\ modid : spec\ \mathbf{end}$	
$module \in \text{Module}$	$::= \mathbf{interface}\ dec$	

Figure 6: Full grammar: Declarations

Derived Form Equivalent Form

Type Variable Sequences *tyvarseq*

ε	$()$
<i>tyvar</i>	$(tyvar)$

Types *ty*

<i>longtycon</i>	$() \text{ longtycon}$
------------------	------------------------

Type Sequences *tyseq*

<i>ty</i>	(ty)
-----------	--------

Figure 7: Derived forms of Type Variable Sequences, Types, and Type Sequences

Derived Form Equivalent Form

Patterns *pat*

<i>var</i>	<i>var as</i> $_$
$pat_1 :: pat_2$	$\text{RML.cons}(pat_1, pat_2)$
$[]$	RML.nil
$[pat_1, \dots, pat_n]$	$pat_1 :: \dots :: pat_n :: []$ ($n \geq 1$)
(pat)	<i>pat</i>

Pattern Sequences *patseq*

<i>pat</i>	(pat)
------------	---------

Pattern Sequences $\langle patseq \rangle$

ε	$()$
---------------	------

Figure 8: Derived forms of Patterns and Pattern Sequences

Derived Form	Equivalent Form
Expressions <i>exp</i>	
$exp_1 :: exp_2$	<code>RML.cons(exp_1, exp_2)</code>
<code>[]</code>	<code>RML.nil</code>
$[exp_1, \dots, exp_n]$	$exp_1 :: \dots :: exp_n :: []$ ($n \geq 1$)
(exp)	exp
Expression Sequences <i>expseq</i>	
exp	(exp)
Expression Sequences $\langle expseq \rangle$	
ε	$()$

Figure 9: Derived forms of Expressions and Expression Sequences

Derived Form	Equivalent Form
Goals <i>goal</i>	
$longvar \langle expseq \rangle$	$longvar \langle expseq \rangle => ()$
$(goal)$	$goal$
Clauses <i>clause</i>	
rule $\langle goal \rangle -- var \langle patseq \rangle$	rule $\langle goal \rangle -- var \langle patseq \rangle => ()$
axiom $var \langle patseq \rangle \langle => result \rangle$	rule $-- var \langle patseq \rangle \langle => result \rangle$
Rules <i>rule</i> Clauses <i>clause</i>	
$clause_1$ default $clause_2$	$clause_1$ $clause_2$

Figure 10: Derived forms of Goals, Clauses, and Rules

Derived Form	Equivalent Form
Specifications <i>spec</i>	
with $scon$	with $interface$
Declarations <i>dec</i>	
with $scon$	with $interface$

Figure 11: Derived forms of Specifications and Declarations

$tyvarseq \in \text{TyVarSeq} ::= (tyvar_1, \dots, tyvar_n)$	$n \geq 0$
$ty \in \text{Ty} ::= tyvar$	variable
$ tyseq \text{ longtycon}$	construction
$ ty_1 * \dots * ty_n$	tuple, $n \geq 2$
$ tyseq_1 \Rightarrow tyseq_2$	relation
$tyseq \in \text{TySeq} ::= (ty_1, \dots, ty_n)$	sequence, $n \geq 0$

Figure 12: Core grammar: Types

$pat \in \text{Pat} ::= _$	wildcard
$ lit$	literal
$ longcon$	constant
$ longcon \text{ patseq}$	structure
$ (pat_1, \dots, pat_n)$	tuple, $n \neq 1$
$ var \text{ as } pat$	binding
$patseq \in \text{PatSeq} ::= (pat_1, \dots, pat_n)$	sequence, $n \geq 0$

Figure 13: Core grammar: Patterns

$exp \in \text{Exp} ::= lit$	literal
$ longcon$	constant
$ longvar$	variable
$ longcon \text{ expseq}$	structure
$ (exp_1, \dots, exp_n)$	tuple, $n \neq 1$
$expseq \in \text{ExpSeq} ::= (exp_1, \dots, exp_n)$	sequence, $n \geq 0$

Figure 14: Core grammar: Expressions

$goal \in \text{Goal} ::= longvar \text{ expseq} \Rightarrow patseq$	call
$ var = exp$	equality
$ \text{let } pat = exp$	binding
$ \text{not } goal$	negation
$ goal_1 \ \& \ goal_2$	sequence
$result \in \text{Result} ::= expseq$	return
$ \text{fail}$	fail
$clause \in \text{Clause} ::= \text{rule } \langle goal \rangle \text{ -- } var \text{ patseq} \Rightarrow result$	rule
$ clause_1 \ clause_2$	sequence

Figure 15: Core grammar: Goals and Clauses

$conbind \in \text{ConBind}$	$::= con \langle of\ ty_1 * \dots * ty_n \rangle$	$\langle n \geq 1 \rangle$
	$ conbind_1 \mid conbind_2$	
$datbind \in \text{DatBind}$	$::= tyvarseq\ tycon = conbind$	
	$ datbind_1 \text{ and } datbind_2$	
$typbind \in \text{TypBind}$	$::= tyvarseq\ tycon = ty$	
	$ typbind_1 \text{ and } typbind_2$	
$withbind \in \text{WithBind}$	$::= \langle withtype\ typbind \rangle$	
$relbind \in \text{RelBind}$	$::= var \langle : ty \rangle = clause$	
	$ relbind_1 \text{ and } relbind_2$	
$spec \in \text{Spec}$	$::= with\ interface$	
	$ type\ tyvarseq\ tycon$	
	$ eqtype\ tyvarseq\ tycon$	
	$ type\ typbind$	
	$ datatype\ datbind\ withbind$	
	$ val\ var : ty$	
	$ relation\ var : ty$	
	$ spec_1\ spec_2$	
$dec \in \text{Dec}$	$::= with\ interface$	
	$ type\ typbind$	
	$ datatype\ datbind\ withbind$	
	$ val\ var = exp$	
	$ relation\ relbind$	
	$ dec_1\ dec_2$	
$interface \in \text{Interface}$	$::= module\ modid : spec\ end$	
$module \in \text{Module}$	$::= interface\ dec$	

Figure 16: Core grammar: Declarations

$modseq \in \text{ModSeq}$	$::= module$
	$ modseq_1\ modseq_2$

Figure 17: Auxiliary grammar: Programs

5 Reordering Phase

This section describes the dependency analysis and reordering phase that occurs between the parsing and static elaboration phases.

5.1 Background

The type system of RML, like those in many other languages, generally requires the declaration of an object to lexically precede every use of that object. This holds for declared types, global variables, relations, and variables local to some relation. We refer to this as the *define-before-use* rule.

In some cases, however, programs can be easier to read if this rule is not enforced. For example, it can be desirable to write a main routine first, followed by the subroutines used to implement the main routine.

Thus RML programs are subjected to a *reordering phase* after parsing but before type checking². This phase reorders top-level declarations of types, variables, and relations, in such a way that the define-before-use rule holds for the reordered program. The sections below describe the exact rules that define when valid reorderings exist.

5.1.1 Terminology

A directed graph $G = (V, E)$ is defined by a set V (the vertices) and a binary relation $E : V \rightarrow V$ (the edges) as usual. When the vertices are syntactic objects and the edges express dependencies between these objects, we call the graph a *dependency graph*.

A dependency graph consists of a set of strongly connected components. Taking these as equivalence classes over the graph results in a DAG of components. The *preorder components* of a dependency graph is a topologically ordered sequence C_1, \dots, C_n of its components such that no C_j has any dependencies to any C_k ($k > j$).

5.2 Type Declarations

The analysis and reordering of type declarations applies to all type aliases *typbind* and datatypes *datbind*. The declarations in a module's interface are treated separately from those in the module's body.

5.2.1 Dependency Analysis

Let the *context* be the set of all type declarations *typbind* and *datbind* in the part of the module under consideration.

²Haskell [?, Section 4.5.1] is another language with a Hindley-Milner type system and reordering of declarations.

With every type declaration is associated a type constructor $tycon_i$ and a set of *immediate dependencies* Dep_i , defined as follows:

- A declaration $tyvarseq\ tycon_i = ty_i$ defines Dep_i as the set of non-qualified type constructors occurring in ty_i and being declared in the current context. It is an error for $tycon_i$ to occur in Dep_i .
- A declaration $tyvarseq\ tycon_i = conbind_i$ defines Dep_i as the set of non-qualified type constructors occurring in $conbind_i$ and being declared in the current context.

It is an error for any $tycon$ to have more than one declaration in the context.

Finally, $Dep = \{(tycon_i, tycon_k) ; tycon_k \in Dep_i\}$ and T is the set of all type constructors declared in the context.

5.2.2 Reordering

Now consider each of the preorder components C_1, \dots, C_n of the dependency graph (T, Dep) :

- If C_j contains a single type constructor from a *typbind*, then its definition is emitted as a single **type** declaration.
- If C_j contains a number of type constructors, all from *datbind* declarations, then their definitions are collected and emitted as a single recursive **datatype** declaration.
- If C_j contains several type constructors, some from *datbind* and some from *typbind* declarations, then further processing is required.

Let $T_j = T \cap C_j$, $Dep'_i = Dep_i \cap T_j$, $Dep_j = \{(tycon_i, tycon_k) ; tycon_k \in Dep'_i\}$, and consider the dependency graph (T_j, Dep_j) . This graph must not have any cycles (i.e., every strongly connected component must be a singleton). A depth-first traversal of (T_j, Dep_j) defines the order in which the *typbind* declarations are collected.

Finally, the component C_j is emitted as a recursive **datatype** declaration, consisting of the *datbind* declarations for the *datbind* type constructors, and a **withtype** containing the *typbind* declarations ordered as described above.

5.3 Value Declarations

The analysis and reordering of value declarations applies to all **val** and *relbind* declarations in module bodies.

5.3.1 Dependency Analysis

Let the context be the set of all **val** and *relbind* declarations in the module's body.

With every value declaration is associated a variable var_i and a set of immediate dependencies Dep_i , defined as follows:

- A declaration **val** $var_i = exp_i$ defines Dep_i as the set of non-qualified variables occurring in exp_i and being declared in the current context. It is an error for var_i to occur in Dep_i .
- A *relbind* declaration $var_i \langle : ty_i \rangle = clause_i$ defines Dep_i as the set of free non-qualified variables in $clause_i$ and being defined in the current context.

It is an error for any var to have more than one declaration in the context.

Finally, $Dep = \{(var_i, var_k) ; var_k \in Dep_i\}$ and V is the set of all variables declared in the context.

5.3.2 Reordering

Now consider each of the preorder components C_1, \dots, C_n of the dependency graph (V, Dep) :

- If C_j contains a single variable from a **val** declaration, then its definition is emitted as is.
- If C_j contains a number of variables, all from *relbind* declarations, then their definitions are collected and emitted as a single recursive **relation** declaration.
- All other cases are illegal.

5.4 Modules

The declarations in a module's interface are reordered separately from those in the module's body. Within each part, declarations are reordered so that **with**-declarations precede type declarations, who in turn precede value declarations.

6 Static Semantics

This section presents the rules for the static semantics of RML. First the semantic objects involved are defined. Then inference rules (written in the style of Natural Semantics) are given.

6.1 Simple Objects

All semantic objects in the static semantics are built from syntactic identifiers and two further kinds of simple objects: booleans and value binding kinds. In the static semantics, we let α range over TyVar .

$$\begin{aligned} \text{Bool} &= \{\text{true}, \text{false}\} \\ vk \in \text{ValKind} &= \{\text{con}, \text{var}, \text{rel}\} \end{aligned}$$

Figure 18: Simple Semantic Objects

6.2 Compound Objects

The compound objects for the static semantics are shown in Figure ??.

For notational convenience, we identify TyVarSeq with TyVar^* , TySeq with Ty^* , PatSeq with Pat^* , and ExpSeq with Exp^* .

$$\begin{aligned} t \in \text{TyName} &= \text{ModId} \times \text{TyCon} \times \text{Bool} \\ \tau \in \text{Type} &= \text{TyVar} \cup \text{TupleType} \cup \text{RelType} \cup \text{ConsType} \\ \text{TypeSeq} &= \cup_{k \geq 0} \text{Type}^k \\ \text{TupleType} &= \text{TypeSeq} \\ \tau^{(k)} [\Rightarrow] \tau'^{(k')} \in \text{RelType} &= \text{TypeSeq} \times \text{TypeSeq} \\ \tau^{(k)} t \in \text{ConsType} &= \cup_{k \geq 0} (\text{Type}^k \times \text{TyName}) \\ \theta \text{ or } \Lambda \alpha^{(k)}. \tau \in \text{TypeFcn} &= \cup_{k \geq 0} (\text{TyVar}^k \times \text{Type}) \\ \sigma \text{ or } \forall \alpha^{(k)}. \tau \in \text{TypeScheme} &= \cup_{k \geq 0} (\text{TyVar}^k \times \text{Type}) \\ CE \text{ or } VE \in \text{ValEnv} &= (\text{Var} \cup \text{Con}) \xrightarrow{\text{fin}} (\text{ValKind} \times \text{TypeScheme}) \\ (\theta, CE_{\text{opt}}) \in \text{TyStr} &= \text{TypeFcn} \times \text{ValEnv} \\ TE \in \text{TyEnv} &= \text{TyCon} \xrightarrow{\text{fin}} \text{TyStr} \\ ME \in \text{ModEnv} &= \text{ModId} \xrightarrow{\text{fin}} (\text{TyEnv} \times \text{ValEnv}) \end{aligned}$$

Figure 19: Compound Semantic Objects

Type names are constructed from module names and type constructors. This is used to uniquely identify all constructed types, since modules may not be redefined, and type constructors may not have multiple bindings within

modules. The only exception to this rule concerns abstract type specifications. However, their implementations are guaranteed to have type names and type functions compatible with those inferred from their specifications.

Every type name t also possesses a boolean equality attribute, which determines whether or not it admits equality. As usual, the expression $\tau_1 = \tau_2$ asserts the structural equality of the two types, but with the modification that the equality attributes of all type names t are ignored.

Value environments map variables and data constructors to type schemes marked with value binding kinds. The kinds distinguish constructors from non-constructors, since these classes behave differently in patterns. They also distinguish variables arising from relation bindings from other variables.

Since the only polymorphic objects in RML are globally declared (relations, variables, and data constructors), type schemes bind either all or none of their type variables. This subsumes Wright's [?] approach for typing references; hence RML does not use SML-style imperative types.

Type environments map type constructors to type structures: tuples of type functions and optional constructor environments. An absent CE signifies that the type is *abstract*, i.e. it is the skeletal type structure from an abstract type specification. Abstract types *must* be rebound in the module's body, whereas other types *may not* be rebound.

6.3 Type Structures and Type Environments

A type structure $(\theta, CE_{\text{opt}})$ is *well-formed* if either $CE_{\text{opt}} = \varepsilon$, or $CE_{\text{opt}} = \{\}$, or θ is of the form $\Lambda\alpha^*. \alpha^* t$, abbreviated as t . (The latter case arises, with CE_{opt} being a non-empty environment, in **datatype** specifications and declarations.) All type structures occurring in elaborations are assumed to be well-formed.

A type structure (t, CE) is said to *respect equality* if, whenever t admits equality, then for each $CE(\text{con})$ of the form $\forall\alpha^{(k)}. (\tau^* [\Rightarrow] [\alpha^{(k)} t])$, the type function $\Lambda\alpha^{(k)}. \tau^*$ also admits equality. (This ensures that the equality goal $=$ will be applicable to a constructed value (con, v^*) of type $\tau^{(k)} t$ only when it is applicable to the tuple v^* itself, whose type is $\tau^* \{\alpha_i \mapsto \tau_i ; 1 \leq i \leq k\}$.) A type environment TE *respects equality* if all its type structures do so.

Let TE be a type environment, and let T be the set of type names t such that (t, CE) occurs in TE for some $CE \neq \{\}$. Then TE is said to *maximise equality* if (a) TE respects equality, and also (b) if any larger subset of T were to admit equality (without any change in the equality attribute of any type names not in T) then TE would cease to respect equality.

6.4 Initial Static Objects

The initial static environments ME_{init} , TE_{init} , VE_{init} , types τ_{char} , τ_{int} , τ_{real} , τ_{string} , type name t_{list} , and set MutTyName , are defined in Section ??.

6.5 Equality

The notion of equality permeates every aspect of the type discipline. The goal $var = exp$ performs an equality test of the values of var and exp at runtime. Equality is defined for literals, nullary constructors, locations, and values built out of such by tuple formation and constructor application. Equality is *not* defined for relations, or aggregates thereof. However, mutable aggregates (e.g. $\alpha \text{ lvar}$) are represented by their locations in the store, and thus *do* admit equality, even if they reference values that do not.

Every type name t has a boolean equality attribute, which determines whether or not it admits equality. If t admits equality, then a type $\tau^{(k)}t$ also admits equality, but only if every τ_i ($1 \leq i \leq k$) does so. Most built-in types, such as `int` and `'a list`, behave this way. The set `MutTyName` contains the type names for the standard mutable types. When t is in `MutTyName`, then a type $\tau^{(k)}t$ admits equality, even if some τ_i does not. If t does not admit equality, then no type $\tau^{(k)}t$ can do so. This is the case if (a) t is the type name for a `datatype`, some of whose constructors reference types that do not admit equality, or (b) t is the type name from a `type` rather than `eqtype` specification in a module's interface.

To permit polymorphism for relations that require equality types, type variables are partitioned in two sets: those that admit equality (`EtyVar`) and those that do not. When a bound type variable α in a type scheme is instantiated to a type τ , if α is in `EtyVar`, then τ is constrained to admit equality.

When a set of new type names are constructed by a `datatype` declaration, as many as possible are set to admit equality. This is expressed by the *maximises equality* property defined for type environments TE .

6.6 Inference Rules

Types

$$\boxed{tyvars \ \tau = \alpha^*}$$

$$\frac{}{tyvars \ \alpha = [\alpha]} \quad (1)$$

$$\frac{tyvars^* \ \tau^* = \alpha^*}{tyvars \ \tau^* = \alpha^*} \quad (2)$$

$$\frac{tyvars^* \ \tau_1^* = \alpha_1^* \quad tyvars^* \ \tau_2^* = \alpha_2^*}{tyvars \ \tau_1^* [\Rightarrow] \tau_2^* = \alpha_1^* \cup \alpha_2^*} \quad (3)$$

$$\frac{tyvars^* \ \tau^* = \alpha^*}{tyvars \ \tau^* t = \alpha^*} \quad (4)$$

$$\boxed{tyvars^* \tau^* = \alpha^*}$$

$$\frac{tyvars \tau_i = \alpha_i^* \ (1 \leq i \leq k) \quad \bigcup_{1 \leq i \leq k} \alpha_i^* = \alpha'^*}{tyvars^* \tau^{(k)} = \alpha'^*} \quad (5)$$

Comment: For conciseness, we identify finite sets and non-repeating sequences.

$$\boxed{\tau \text{ admits } Eq}$$

$$\frac{\alpha \in \text{EtyVar}}{\alpha \text{ admits } Eq} \quad (6)$$

$$\frac{\tau^* \text{ admits } Eq}{\tau^* \text{ admits } Eq} \quad (7)$$

$$\frac{t \text{ admits equality} \quad t \in \text{MutTyName}}{\tau^* t \text{ admits } Eq} \quad (8)$$

$$\frac{t \text{ admits equality} \quad \tau^* \text{ admits } Eq}{\tau^* t \text{ admits } Eq} \quad (9)$$

Comment: Constructed types whose type names do not admit equality, as well as relation types, never admit equality.

$$\boxed{\tau^* \text{ admits } Eq}$$

$$\frac{\tau_i \text{ admits } Eq \ (1 \leq i \leq k)}{\tau^{(k)} \text{ admits } Eq} \quad (10)$$

Type Functions

$$\boxed{\text{applyTypeFcn}(\theta, \tau^{(k)}) = \tau'}$$

$$\frac{k = k' \quad \tau\{\alpha_i \mapsto \tau_i \ ; \ 1 \leq i \leq k\} = \tau'}{\text{applyTypeFcn}(\Lambda\alpha^{(k)}. \tau, \tau^{(k')}) = \tau'} \quad (11)$$

$$\boxed{\theta \text{ admits } Eq}$$

$$\frac{\tau\{\alpha_i \mapsto \alpha'_i \ ; \ \alpha'_i \in \text{EtyVar}, \ 1 \leq i \leq k\} = \tau' \quad \tau' \text{ admits } Eq}{\Lambda\alpha^{(k)}. \tau \text{ admits } Eq} \quad (12)$$

Comment: When determining if a type function admits equality, the equality status of its bound type variables is ignored.

Type Schemes

$$\boxed{Close \ \tau = \sigma}$$

$$\frac{tyvars \ \tau = \alpha^*}{Close \ \tau = \forall \alpha^*. \tau} \quad (13)$$

Comment: This operation closes τ by abstracting all of its type variables.

$$\boxed{\alpha \succ \tau}$$

$$\frac{\alpha \in \text{EtyVar} \quad \tau \text{ admits } Eq}{\alpha \succ \tau} \quad (14)$$

$$\frac{\alpha \notin \text{EtyVar}}{\alpha \succ \tau} \quad (15)$$

$$\boxed{\sigma \succ \tau}$$

$$\frac{\tau\{\alpha_i \mapsto \tau'_i; \ \alpha_i \succ \tau'_i, \ 1 \leq i \leq k\} = \tau''}{\forall \alpha^{(k)}. \tau \succ \tau''} \quad (16)$$

Comment: This operation instantiates a type scheme by substituting permissible types for the abstracted type variables.

Long Type Constructors

$$\boxed{ME, TE \vdash longtycon \Rightarrow \theta}$$

$$\frac{(TE_{\text{init}} + TE)(tycon) = (\theta, _)}{ME, TE \vdash tycon \Rightarrow \theta} \quad (17)$$

$$\frac{ME(modid) = (TE', _) \quad TE'(tycon) = (\theta, _)}{ME, TE \vdash modid . tycon \Rightarrow \theta} \quad (18)$$

Type Expressions

$$\boxed{ME, TE\langle, \alpha^* \rangle \vdash ty \Rightarrow \tau}$$

$$\frac{\langle \alpha \in \alpha^* \rangle}{ME, TE\langle, \alpha^* \rangle \vdash \alpha \Rightarrow \alpha} \quad (19)$$

$$\frac{ME, TE\langle, \alpha^* \rangle \vdash tyseq \Rightarrow \tau^* \quad ME, TE \vdash longtycon \Rightarrow \theta \quad applyTypeFcn(\theta, \tau^*) = \tau}{ME, TE\langle, \alpha^* \rangle \vdash tyseq \ longtycon \Rightarrow \tau} \quad (20)$$

$$\frac{ME, TE\langle, \alpha^* \rangle \vdash ty^{(k)} \Rightarrow \tau^{(k)}}{ME, TE\langle, \alpha^* \rangle \vdash ty_1 * \dots * ty_k \Rightarrow \tau^{(k)} \text{ in Type}} \quad (21)$$

$$\frac{ME, TE\langle, \alpha^* \rangle \vdash tyseq_1 \Rightarrow \tau_1^* \quad ME, TE\langle, \alpha^* \rangle \vdash tyseq_2 \Rightarrow \tau_2^*}{ME, TE\langle, \alpha^* \rangle \vdash tyseq_1 \Rightarrow tyseq_2 \Rightarrow \tau_1^* [\Rightarrow] \tau_2^*} \quad (22)$$

Comment: ?? When present, α^* serves to constrain type expressions to only have type variables in α^* . This is used when elaborating *typbinds* and *datbinds*. It does *not* suffice to elaborate a type expression to a type τ , and then check that *tyvars* $\tau = \alpha'^*$ and $\alpha'^* \subseteq \alpha^*$, since type functions θ may ‘forget’ some of their type variables. The following example does not elaborate: `type 'a t = int type tt = 'b t`.

Type Expression Sequences

$$\boxed{ME, TE\langle, \alpha^* \rangle \vdash ty^{(k)} \Rightarrow \tau^{(k)}}$$

$$\frac{ME, TE\langle, \alpha^* \rangle \vdash ty_i \Rightarrow \tau_i \quad (1 \leq i \leq k)}{ME, TE\langle, \alpha^* \rangle \vdash ty^{(k)} \Rightarrow \tau^{(k)}} \quad (23)$$

Type Variable Sequences

$$\boxed{nodups \alpha^*}$$

$$\overline{nodups \quad \square} \quad (24)$$

$$\frac{\alpha \notin \alpha^* \quad nodups \alpha^*}{nodups \alpha :: \alpha^*} \quad (25)$$

Comment: *nodups* α^* verifies that α^* contains no duplicates, as required when elaborating type specifications and declarations.

Type Bindings

$$\boxed{ME, TE, TE_1 \vdash typbind \Rightarrow TE_2}$$

$$\frac{tycon \notin \text{Dom } TE \quad tycon \notin \text{Dom } TE_1 \quad nodups \alpha^* \quad ME, TE + TE_1, \alpha^* \vdash ty \Rightarrow \tau \quad \theta = \Lambda \alpha^*. \tau}{ME, TE, TE_1 \vdash \alpha^* \quad tycon = ty \Rightarrow TE_1 + \{tycon \mapsto (\theta, \{\})\}} \quad (26)$$

$$\frac{ME, TE, TE_1 \vdash typbind_1 \Rightarrow TE_2 \quad ME, TE, TE_2 \vdash typbind_2 \Rightarrow TE_3}{ME, TE, TE_1 \vdash typbind_1 \text{ and } typbind_2 \Rightarrow TE_3} \quad (27)$$

$$\boxed{ME, TE \vdash withbind \Rightarrow TE}$$

$$\overline{ME, TE \vdash \varepsilon \Rightarrow \{\}} \quad (28)$$

$$\frac{ME, TE, \{\} \vdash \text{typbind} \Rightarrow TE'}{ME, TE \vdash \text{withtype } \text{typbind} \Rightarrow TE'} \quad (29)$$

Comment: As implied by the sequential elaboration of a *typbind*, the type aliases after **withtype** are expanded sequentially. This is intensional.

Datatype Bindings

$$\boxed{TE, TE_{\text{skel}}, \text{modid} \vdash \text{datbind} \Rightarrow TE'_{\text{skel}}}$$

$$\frac{\begin{array}{l} (tycon \notin \text{Dom } TE \vee TE(tycon) = (_, \varepsilon)) \quad tycon \notin \text{Dom } TE_{\text{skel}} \\ nodups \ \alpha^* \quad t = (\text{modid}, tycon, eq) \quad \theta = \Lambda \alpha^*. \alpha^* t \end{array}}{TE, TE_{\text{skel}}, \text{modid} \vdash \alpha^* tycon = _ \Rightarrow TE_{\text{skel}} + \{tycon \mapsto (\theta, \{\})\}} \quad (30)$$

$$\frac{\begin{array}{l} TE, TE_{\text{skel}}^1, \text{modid} \vdash \text{datbind}_1 \Rightarrow TE_{\text{skel}}^2 \\ TE, TE_{\text{skel}}^2, \text{modid} \vdash \text{datbind}_2 \Rightarrow TE_{\text{skel}}^3 \end{array}}{TE, TE_{\text{skel}}^1, \text{modid} \vdash \text{datbind}_1 \text{ and } \text{datbind}_2 \Rightarrow TE_{\text{skel}}^3} \quad (31)$$

Comments:

?? Normally, once an identifier has been bound in an environment, it may not be rebound. This is the only place where this does not hold: an abstract type declared in an interface can (indeed must) be rebound in the module's body by a *datbind*. This relies critically on the fact that the *datbind* will produce a compatible type name and type function. See also rules ??, ??, ??, and ??.

?? The equality attribute of the new type name is determined by the *maximises equality* condition in rule ??.

Constructor Bindings

$$\boxed{ME, TE, VE, CE, \alpha^*, \tau \vdash \text{conbind} \Rightarrow CE'}$$

$$\frac{\text{con} \notin \text{Dom } VE \quad \text{con} \notin \text{Dom } CE \quad \text{Close } \tau = \sigma}{ME, TE, VE, CE, \alpha^*, \tau \vdash \text{con} \Rightarrow CE + \{\text{con} \mapsto (\text{con}, \sigma)\}} \quad (32)$$

$$\frac{\begin{array}{l} \text{con} \notin \text{Dom } VE \quad \text{con} \notin \text{Dom } CE \quad ME, TE, \alpha^* \vdash ty^{(k)} \Rightarrow \tau'^{(k)} \\ \text{Close } \tau'^{(k)} [\Rightarrow] [\tau] = \sigma \quad CE' = CE + \{\text{con} \mapsto (\text{con}, \sigma)\} \end{array}}{ME, TE, VE, CE, \alpha^*, \tau \vdash \text{con of } ty_1 * \dots * ty_k \Rightarrow CE'} \quad (33)$$

$$\frac{\begin{array}{l} ME, TE, VE, CE, \alpha^*, \tau \vdash \text{conbind}_1 \Rightarrow CE_1 \\ ME, TE, VE, CE_1, \alpha^*, \tau \vdash \text{conbind}_2 \Rightarrow CE_2 \end{array}}{ME, TE, VE, CE, \alpha^*, \tau \vdash \text{conbind}_1 \mid \text{conbind}_2 \Rightarrow CE_2} \quad (34)$$

$$\boxed{ME, TE, VE \vdash \text{datbind} \Rightarrow TE', VE'}$$

$$\frac{\begin{array}{l} TE(\text{tycon}) = (\theta, _) \quad \text{applyTypeFcn}(\theta, \alpha^*) = \tau \\ ME, TE, VE, \{\}, \alpha^*, \tau \vdash \text{conbind} \Rightarrow CE \quad VE' = VE + CE \end{array}}{ME, TE, VE \vdash \alpha^* \text{ tycon} = \text{conbind} \Rightarrow \{\text{tycon} \mapsto (\theta, CE)\}, VE'} \quad (35)$$

$$\frac{\begin{array}{l} ME, TE, VE \vdash \text{datbind}_1 \Rightarrow TE_1, VE_1 \\ ME, TE, VE_1 \vdash \text{datbind}_2 \Rightarrow TE_2, VE_2 \end{array}}{ME, TE, VE \vdash \text{datbind}_1 \text{ and } \text{datbind}_2 \Rightarrow TE_1 + TE_2, VE_2} \quad (36)$$

$$\boxed{ME, TE, VE, \text{modid} \vdash \text{datbind}, \text{withbind} \Rightarrow TE, VE}$$

$$\frac{\begin{array}{l} TE, \{\}, \text{modid} \vdash \text{datbind} \Rightarrow TE_{\text{skel}} \\ ME, TE + TE_{\text{skel}} \vdash \text{withbind} \Rightarrow TE_{\text{with}} \\ ME, TE + TE_{\text{skel}} + TE_{\text{with}}, VE \vdash \text{datbind} \Rightarrow TE_{\text{data}}, VE' \\ TE_{\text{data}} \text{ maximises equality} \\ TE' = TE + TE_{\text{data}} + TE_{\text{with}} \end{array}}{ME, TE, VE, \text{modid} \vdash \text{datbind}, \text{withbind} \Rightarrow TE', VE'} \quad (37)$$

Variables

$$\boxed{VE_{\text{outer}}, VE_{\text{inner}} \vdash \text{var}}$$

$$\frac{\begin{array}{l} \text{var} \notin \text{Dom } VE_{\text{inner}} \\ (\text{var} \notin \text{Dom } VE_{\text{outer}} \vee VE_{\text{outer}}(\text{var}) \neq (\text{con}, _)) \\ (\text{var} \notin \text{Dom } VE_{\text{init}} \vee VE_{\text{init}}(\text{var}) \neq (\text{con}, _)) \end{array}}{VE_{\text{outer}}, VE_{\text{inner}} \vdash \text{var}} \quad (38)$$

Comment: This rule checks that a value identifier may be bound as a variable, at a point where the visible value environment is $VE_{\text{init}} + VE_{\text{outer}} + VE_{\text{inner}}$. The first premiss prevents multiple bindings of the same identifier in a given “scope.” The following premisses ensure that an identifier cannot be bound as a variable if it already has a visible binding as a value constructor.

Specifications

$$\boxed{ME, TE, VE, \text{modid} \vdash \text{spec} \Rightarrow ME, TE, VE}$$

$$\frac{\begin{array}{l} ME \vdash \text{interface} \Rightarrow _, TE', VE', \text{modid}' \\ ME' = ME + \{\text{modid}' \mapsto (TE', VE')\} \end{array}}{ME, TE, VE, \text{modid} \vdash \text{with interface} \Rightarrow ME', TE, VE} \quad (39)$$

$$\frac{\begin{array}{l} \text{tycon} \notin \text{Dom } TE \quad \text{nodups } \alpha^* \quad t = (\text{modid}, \text{tycon}, \text{false}) \\ TE' = TE + \{\text{tycon} \mapsto (\Lambda \alpha^*. \alpha^* t, \varepsilon)\} \end{array}}{ME, TE, VE, \text{modid} \vdash \text{type } \alpha^* \text{ tycon} \Rightarrow ME, TE', VE} \quad (40)$$

$$\frac{\begin{array}{c} tycon \notin \text{Dom } TE \quad \text{nodups } \alpha^* \quad t = (modid, tycon, \text{true}) \\ TE' = TE + \{tycon \mapsto (\Lambda \alpha^*. \alpha^* t, \varepsilon)\} \end{array}}{ME, TE, VE, modid \vdash \mathbf{eqtype} \alpha^* tycon \Rightarrow ME, TE', VE} \quad (41)$$

$$\frac{ME, \{\}, TE \vdash \mathbf{typbind} \Rightarrow TE'}{ME, TE, VE, modid \vdash \mathbf{type} \mathbf{typbind} \Rightarrow ME, TE', VE} \quad (42)$$

$$\frac{ME, TE, VE, modid \vdash \mathbf{datbind}, \mathbf{withbind} \Rightarrow TE', VE'}{ME, TE, VE, modid \vdash \mathbf{datatype} \mathbf{datbind} \mathbf{withbind} \Rightarrow ME, TE', VE'} \quad (43)$$

$$\frac{\begin{array}{c} \{\}, VE \vdash \mathbf{var} \quad ME, TE \vdash \mathbf{ty} \Rightarrow \tau \\ \text{Close } \tau = \sigma \quad VE' = VE + \{\mathbf{var} \mapsto (\mathbf{var}, \sigma)\} \end{array}}{ME, TE, VE, modid \vdash \mathbf{val} \mathbf{var} : \mathbf{ty} \Rightarrow ME, TE, VE'} \quad (44)$$

$$\frac{\begin{array}{c} \{\}, VE \vdash \mathbf{var} \quad ME, TE \vdash \mathbf{ty} \Rightarrow \tau \\ \tau = \tau_1^* [\Rightarrow] \tau_2^* \quad \text{Close } \tau = \sigma \quad VE' = VE + \{\mathbf{var} \mapsto (\mathbf{rel}, \sigma)\} \end{array}}{ME, TE, VE, modid \vdash \mathbf{relation} \mathbf{var} : \mathbf{ty} \Rightarrow ME, TE, VE'} \quad (45)$$

$$\frac{\begin{array}{c} ME, TE, VE, modid \vdash \mathbf{spec}_1 \Rightarrow ME_1, TE_1, VE_1 \\ ME_1, TE_1, VE_1, modid \vdash \mathbf{spec}_2 \Rightarrow ME_2, TE_2, VE_2 \end{array}}{ME, TE, VE, modid \vdash \mathbf{spec}_1 \mathbf{spec}_2 \Rightarrow ME_2, TE_2, VE_2} \quad (46)$$

Interfaces

$$\boxed{ME \vdash \mathbf{interface} \Rightarrow ME, TE, VE, modid}$$

$$\frac{modid \notin \text{Dom } ME \quad ME_{\text{init}}, \{\}, \{\}, modid \vdash \mathbf{spec} \Rightarrow ME', TE, VE}{ME \vdash \mathbf{module} \mathbf{modid} : \mathbf{spec} \mathbf{end} \Rightarrow ME', TE, VE, modid} \quad (47)$$

Literals

$$\boxed{\mathbf{lit} \Rightarrow \tau}$$

$$\frac{}{ccon \Rightarrow \tau_{\text{char}}} \quad (48)$$

$$\frac{}{icon \Rightarrow \tau_{\text{int}}} \quad (49)$$

$$\frac{}{rcon \Rightarrow \tau_{\text{real}}} \quad (50)$$

$$\frac{}{scon \Rightarrow \tau_{\text{string}}} \quad (51)$$

Long Value Constructors

$$\boxed{ME, VE \vdash longcon \Rightarrow \sigma}$$

$$\frac{(VE_{\text{init}} + VE)(con) = (con, \sigma)}{ME, VE \vdash con \Rightarrow \sigma} \quad (52)$$

$$\frac{ME(modid) = (_, VE') \quad VE'(con) = (con, \sigma)}{ME, VE \vdash modid . con \Rightarrow \sigma} \quad (53)$$

Patterns

$$\boxed{ME, VE, VE_{\text{pat}} \vdash pat \Rightarrow VE'_{\text{pat}}, \tau}$$

$$\frac{}{ME, VE, VE_{\text{pat}} \vdash _ \Rightarrow VE_{\text{pat}}, \tau} \quad (54)$$

$$\frac{lit \Rightarrow \tau}{ME, VE, VE_{\text{pat}} \vdash lit \Rightarrow VE_{\text{pat}}, \tau} \quad (55)$$

$$\frac{ME, VE \vdash longcon \Rightarrow \sigma \quad \sigma \succ \tau^* t}{ME, VE, VE_{\text{pat}} \vdash longcon \Rightarrow VE_{\text{pat}}, \tau^* t} \quad (56)$$

$$\frac{ME, VE \vdash longcon \Rightarrow \sigma \quad ME, VE, VE_{\text{pat}} \vdash patseq \Rightarrow VE'_{\text{pat}}, \tau^* \quad \sigma \succ \tau^* [\Rightarrow][\tau]}{ME, VE, VE_{\text{pat}} \vdash longcon \quad patseq \Rightarrow VE'_{\text{pat}}, \tau} \quad (57)$$

$$\frac{ME, VE, VE_{\text{pat}} \vdash pat^{(k)} \Rightarrow VE'_{\text{pat}}, \tau^{(k)}}{ME, VE, VE_{\text{pat}} \vdash (pat_1, \dots, pat_k) \Rightarrow VE'_{\text{pat}}, \tau^{(k)} \text{ in Type}} \quad (58)$$

$$\frac{ME, VE, VE_{\text{pat}} \vdash pat \Rightarrow VE'_{\text{pat}}, \tau \quad VE, VE'_{\text{pat}} \vdash var}{ME, VE, VE_{\text{pat}} \vdash var \text{ as } pat \Rightarrow VE'_{\text{pat}} + \{var \mapsto (var, \forall[].\tau)\}, \tau} \quad (59)$$

Comment: VE_{pat} is the environment being built for the current pattern, while VE is the environment in which this is taking place. This split is used to allow pattern variables to shadow previous bindings.

Pattern Sequences

$$\boxed{ME, VE, VE_{\text{pat}} \vdash pat^{(k)} \Rightarrow VE'_{\text{pat}}, \tau^{(k)}}$$

$$\frac{ME, VE, VE_{\text{pat}}^i \vdash pat_i \Rightarrow VE_{\text{pat}}^{i+1}, \tau_i \quad (1 \leq i \leq k)}{ME, VE, VE_{\text{pat}}^1 \vdash pat^{(k)} \Rightarrow VE_{\text{pat}}^{k+1}, \tau^{(k)}} \quad (60)$$

Long Variables

$$\boxed{ME, VE \vdash longvar \Rightarrow \sigma}$$

$$\frac{(VE_{\text{init}} + VE)(var) = (vk, \sigma) \quad vk \neq con}{ME, VE \vdash var \Rightarrow \sigma} \quad (61)$$

$$\frac{ME(modid) = (_, VE') \quad VE'(var) = (vk, \sigma) \quad vk \neq \text{con}}{ME, VE \vdash modid.var \Rightarrow \sigma} \quad (62)$$

Expressions

$$\boxed{ME, VE \vdash exp \Rightarrow \tau}$$

$$\frac{lit \Rightarrow \tau}{ME, VE \vdash lit \Rightarrow \tau} \quad (63)$$

$$\frac{ME, VE \vdash longcon \Rightarrow \sigma \quad \sigma \succ \tau^* t}{ME, VE \vdash longcon \Rightarrow \tau^* t} \quad (64)$$

$$\frac{ME, VE \vdash longvar \Rightarrow \sigma \quad \sigma \succ \tau}{ME, VE \vdash longvar \Rightarrow \tau} \quad (65)$$

$$\frac{ME, VE \vdash longcon \Rightarrow \sigma \quad ME, VE \vdash expseq \Rightarrow \tau^* \quad \sigma \succ \tau^* [\Rightarrow][\tau]}{ME, VE \vdash longcon \expseq \Rightarrow \tau} \quad (66)$$

$$\frac{ME, VE \vdash exp^{(k)} \Rightarrow \tau^{(k)}}{ME, VE \vdash (exp_1, \dots, exp_k) \Rightarrow \tau^{(k)} \text{ in Type}} \quad (67)$$

Comment: ?? Non-constant data constructors are excluded here.

Expression Sequences

$$\boxed{ME, VE \vdash exp^{(k)} \Rightarrow \tau^{(k)}}$$

$$\frac{ME, VE \vdash exp_i \Rightarrow \tau_i \quad (1 \leq i \leq k)}{ME, VE \vdash exp^{(k)} \Rightarrow \tau^{(k)}} \quad (68)$$

Goals

$$\boxed{ME, VE \vdash goal \Rightarrow VE}$$

$$\frac{ME, VE \vdash longvar \Rightarrow \sigma \quad ME, VE \vdash expseq \Rightarrow \tau_1^* \quad ME, VE, \{\} \vdash patseq \Rightarrow VE_{\text{pat}}, \tau_2^* \quad \sigma \succ \tau_1^* [\Rightarrow][\tau_2^*]}{ME, VE \vdash longvar \expseq \Rightarrow patseq \Rightarrow VE + VE_{\text{pat}}} \quad (69)$$

$$\frac{ME, VE \vdash exp \Rightarrow \tau \quad VE(var) = (_, \sigma) \quad \sigma \succ \tau \quad \tau \text{ admits } Eq}{ME, VE \vdash var = exp \Rightarrow VE} \quad (70)$$

$$\frac{ME, VE \vdash exp \Rightarrow \tau \quad ME, VE, \{\} \vdash pat \Rightarrow VE_{\text{pat}}, \tau' \quad \tau = \tau'}{ME, VE \vdash \text{let } pat = exp \Rightarrow VE + VE_{\text{pat}}} \quad (71)$$

$$\frac{ME, VE \vdash goal \Rightarrow VE'}{ME, VE \vdash \text{not } goal \Rightarrow VE} \quad (72)$$

$$\frac{ME, VE \vdash goal_1 \Rightarrow VE_1 \quad ME, VE_1 \vdash goal_2 \Rightarrow VE_2}{ME, VE \vdash goal_1 \& goal_2 \Rightarrow VE_2} \quad (73)$$

Comment: ?? A negative goal produces no visible bindings.

$$\boxed{ME, VE \vdash' \langle goal \rangle \Rightarrow VE}$$

$$\frac{ME, VE \vdash goal \Rightarrow VE'}{ME, VE \vdash' goal \Rightarrow VE'} \quad (74)$$

$$\frac{}{ME, VE \vdash' \varepsilon \Rightarrow VE} \quad (75)$$

Results

$$\boxed{ME, VE \vdash result \Rightarrow \tau^*}$$

$$\frac{ME, VE \vdash expseq \Rightarrow \tau^*}{ME, VE \vdash expseq \Rightarrow \tau^*} \quad (76)$$

$$\frac{}{ME, VE \vdash \mathbf{fail} \Rightarrow \tau^*} \quad (77)$$

Comment: ?? The type sequence τ^* is determined by the other clauses in the relation being checked.

Clauses

$$\boxed{ME, VE, var_{\text{rel}}, \tau \vdash clause}$$

$$\frac{ME, VE, \{\} \vdash patseq \Rightarrow VE_{\text{pat}}, \tau_1^* \quad ME, VE + VE_{\text{pat}} \vdash' \langle goal \rangle \Rightarrow VE' \quad ME, VE' \vdash result \Rightarrow \tau_2^* \quad \tau = \tau_1^* [\Rightarrow] \tau_2^* \quad var_{\text{rel}} = var}{ME, VE, var_{\text{rel}}, \tau \vdash \mathbf{rule} \langle goal \rangle \text{ -- } var \text{ patseq } \Rightarrow result} \quad (78)$$

$$\frac{ME, VE, var_{\text{rel}}, \tau \vdash clause_1 \quad ME, VE, var_{\text{rel}}, \tau \vdash clause_2}{ME, VE, var_{\text{rel}}, \tau \vdash clause_1 \text{ clause}_2} \quad (79)$$

Relation Bindings

$$\boxed{ME, VE \vdash relbind}$$

$$\frac{VE(var) = (\text{rel}, \forall \tau) \quad ME, VE, var, \tau \vdash clause}{ME, VE \vdash var \langle : ty \rangle = clause} \quad (80)$$

$$\frac{ME, VE \vdash relbind_1 \quad ME, VE \vdash relbind_2}{ME, VE \vdash relbind_1 \text{ and } relbind_2} \quad (81)$$

$$\boxed{ME, TE, VE, VE_{\text{rel}} \vdash \text{relbind} \Rightarrow VE'_{\text{rel}}}$$

$$\frac{\{\}, VE + VE_{\text{rel}} \vdash \text{var} \quad \langle ME, TE \vdash ty \Rightarrow \tau \rangle \quad \sigma = \forall[].\tau}{ME, TE, VE, VE_{\text{rel}} \vdash \text{var} \quad \langle : ty \rangle = _ \Rightarrow VE_{\text{rel}} + \{\text{var} \mapsto (\text{rel}, \sigma)\}} \quad (82)$$

$$\frac{ME, TE, VE, VE_{\text{rel}}^1 \vdash \text{relbind}_1 \Rightarrow VE_{\text{rel}}^2 \quad ME, TE, VE, VE_{\text{rel}}^2 \vdash \text{relbind}_2 \Rightarrow VE_{\text{rel}}^3}{ME, TE, VE, VE_{\text{rel}}^1 \vdash \text{relbind}_1 \text{ and } \text{relbind}_2 \Rightarrow VE_{\text{rel}}^3} \quad (83)$$

Variable Environments

$$\boxed{\text{Close } VE = VE'}$$

$$\frac{VE' = \{\text{var}_i \mapsto (\text{rel}, \sigma_i) ; \text{Close } \tau_i = \sigma_i, 1 \leq i \leq k\}}{\text{Close } \{\text{var}_i \mapsto (_, \forall_.\tau_i) ; 1 \leq i \leq k\} = VE'} \quad (84)$$

Declarations

$$\boxed{ME, TE, VE, \text{modid} \vdash \text{dec} \Rightarrow ME, TE, VE}$$

$$\frac{ME \vdash \text{interface} \Rightarrow _, TE', VE', \text{modid}' \quad ME' = ME + \{\text{modid}' \mapsto (TE', VE')\}}{ME, TE, VE, \text{modid} \vdash \text{with interface} \Rightarrow ME', TE, VE} \quad (85)$$

$$\frac{ME, \{\}, TE \vdash \text{typbind} \Rightarrow TE'}{ME, TE, VE, \text{modid} \vdash \text{type typbind} \Rightarrow ME, TE', VE} \quad (86)$$

$$\frac{ME, TE, VE, \text{modid} \vdash \text{datbind}, \text{withbind} \Rightarrow TE', VE'}{ME, TE, VE, \text{modid} \vdash \text{datatype datbind withbind} \Rightarrow ME, TE', VE'} \quad (87)$$

$$\frac{\{\}, VE \vdash \text{var} \quad ME, VE \vdash \text{exp} \Rightarrow \tau \quad \text{Close } \tau = \sigma \quad VE' = VE + \{\text{var} \mapsto (\text{var}, \sigma)\}}{ME, TE, VE, \text{modid} \vdash \text{val var} = \text{exp} \Rightarrow ME, TE, VE'} \quad (88)$$

$$\frac{ME, TE, VE, \{\} \vdash \text{relbind} \Rightarrow VE_{\text{rel}} \quad ME, VE + VE_{\text{rel}} \vdash \text{relbind} \quad \text{Close } VE_{\text{rel}} = VE'_{\text{rel}}}{ME, TE, VE, \text{modid} \vdash \text{relation relbind} \Rightarrow ME, TE, VE + VE'_{\text{rel}}} \quad (89)$$

$$\frac{ME, TE, VE, \text{modid} \vdash \text{dec}_1 \Rightarrow ME_1, TE_1, VE_1 \quad ME_1, TE_1, VE_1, \text{modid} \vdash \text{dec}_2 \Rightarrow ME_2, TE_2, VE_2}{ME, TE, VE, \text{modid} \vdash \text{dec}_1 \text{ dec}_2 \Rightarrow ME_2, TE_2, VE_2} \quad (90)$$

Specifications

$$\boxed{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash \text{spec}}$$

$$\frac{VE_{\text{spec}}(\text{var}) = (\text{var}, \forall_.\tau_{\text{spec}}) \quad VE_{\text{dec}}(\text{var}) = (_, \sigma_{\text{dec}}) \quad \sigma_{\text{dec}} \succ \tau_{\text{spec}}}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash \text{val var} : _} \quad (91)$$

$$\frac{VE_{\text{spec}}(var) = (\text{rel}, \forall _ . \tau_{\text{spec}}) \quad VE_{\text{dec}}(var) = (\text{rel}, \sigma_{\text{dec}}) \quad \sigma_{\text{dec}} \succ \tau_{\text{spec}}}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash \mathbf{relation} \, var : _} \quad (92)$$

$$\frac{TE_{\text{dec}}(tycon) = (\Lambda \alpha^* . \tau, CE) \quad \# \alpha^* = \# \alpha'^*}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash \mathbf{type} \, \alpha'^* \, tycon} \quad (93)$$

$$\frac{TE_{\text{dec}}(tycon) = (\Lambda \alpha^* . \tau, CE) \quad \# \alpha^* = \# \alpha'^* \quad \Lambda \alpha^* . \tau \text{ admitsEq}}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash \mathbf{eqtype} \, \alpha'^* \, tycon} \quad (94)$$

$$\frac{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash spec_1 \quad TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash spec_2}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash spec_1 \, spec_2} \quad (95)$$

$$\frac{spec = \mathbf{with} \, _ \vee spec = \mathbf{type} \, typbind \vee spec = \mathbf{datatype} \, _}{TE_{\text{dec}}, VE_{\text{spec}}, VE_{\text{dec}} \vdash spec} \quad (96)$$

Comments:

?? ?? Note that σ_{dec} cannot be more specific than τ_{spec} .

?? ?? Note that CE must be present.

Modules

$$\boxed{module \Rightarrow modid, VE}$$

$$\frac{\begin{array}{l} ME_{\text{init}} \vdash \mathbf{module} \, modid : spec \, \mathbf{end} \Rightarrow ME, TE, VE, modid \\ VE' = \{con \mapsto VE(con) ; VE(con) = (con, _)\} \\ ME, TE, VE', modid \vdash dec \Rightarrow ME', TE', VE'' \quad TE', VE, VE'' \vdash spec \end{array}}{\mathbf{module} \, modid : spec \, \mathbf{end} \, dec \Rightarrow modid, VE} \quad (97)$$

Comment: The VE arising from the elaboration of the interface contains bindings for both constructors and non-constructors. Only the constructor bindings are retained (in VE') when the module body is elaborated.

Module Sequences

$$\boxed{ME \vdash modseq \Rightarrow ME'}$$

$$\frac{module \Rightarrow modid, VE \quad modid \notin \text{Dom } ME}{ME \vdash module \Rightarrow ME + \{modid \mapsto (\{\}, VE)\}} \quad (98)$$

$$\frac{ME \vdash modseq_1 \Rightarrow ME_1 \quad ME_1 \vdash modseq_2 \Rightarrow ME_2}{ME \vdash modseq_1 \, modseq_2 \Rightarrow ME_2} \quad (99)$$

Programs

$$\boxed{\vdash \text{modseq}}$$

$$\frac{ME_{\text{init}} \vdash \text{modseq} \Rightarrow ME \quad ME(\mathbf{Main}) = (_, VE) \quad VE(\mathbf{main}) = (\text{rel}, \sigma) \quad \sigma \succ [[\tau_{\text{string}}]t_{\text{list}}] \Rightarrow []}{\vdash \text{modseq}} \quad (100)$$

Comment: A program is a collection of modules. The program's entry point is module **Main**'s relation **main**, which must have type **string list => ()**.

7 Dynamic Semantics

7.1 Simple Objects

All objects in the dynamic semantics are built from syntactic objects and the object classes shown in Figure ??.

$a \in \text{Answer} = \{\text{Yes}, \text{No}\}$	final answers
$l \in \text{Loc}$	denumerable set of locations
$\text{prim} \in \text{PrimVal}$	primitive procedures and values
$\{\text{FAIL}\}$	failure token

Figure 20: Simple Semantic Objects

7.2 Compound Objects

The compound objects for the dynamic semantics are shown in Figures ?? and ??.

$v \in \text{Val} = \text{Lit} \cup \text{PrimVal} \cup \text{Val}^* \cup (\text{Con} \times \text{Val}^*) \cup \text{Closure} \cup \text{Loc}$
$\text{Closure} = \text{Clause} \times \text{MEnv} \times \text{VEnv} \times \text{VEnv}$
$VE \in \text{VEnv} = \text{Var} \xrightarrow{\text{fin}} \text{Val}$
$ME \in \text{MEnv} = \text{ModId} \xrightarrow{\text{fin}} \text{VEnv}$
$\sigma \in \text{Store} = \text{Loc} \xrightarrow{\text{fin}} (\text{Val} \cup \{\text{unbound}\})$
$s \in \text{State} = \text{Store} \times \dots$
$m \in \text{Marker} = \text{Store}$

Figure 21: Compound Semantic Objects

$fc \in \text{FCont} ::= \text{orhalt}$	failure continuations
$\text{orelse}(m, \text{clause}, v^*, ME, VE, fc, pc)$	
$\text{ornot}(m, gc, VE, fc)$	
$gc \in \text{GCont} ::= \text{andhalt}$	goal continuations
$\text{andthen}(\text{goal}, ME, gc)$	
$\text{andnot}(fc)$	
$\text{andreturn}(ME, \text{result}, pc)$	
$pc \in \text{PCont} ::= \text{retmatch}(pat^*, VE, gc, fc)$	procedure continuations

Figure 22: Grammar of Continuation Terms

Expressible values are literals, primitive procedures and values, tuples, constructor applications, closures, or locations.

Closures are clauses closed with the module and variable environments in effect when the closure (resulting from evaluating a *relbind*) was created. The last component of a closure describes the recursive part of the closure's environment. This component will be successively unfolded during recursions, giving each level of the recursion access to a 'virtually recursive' environment without actually creating a recursive object.

Logical variables behave like write-once references. A logical variable of type α *lvar* is represented by a location, and the store maps that location either to a value (when the variable has been instantiated) or to the token 'unbound' (before it has been instantiated).

Markers are used to record whatever information is necessary to restore a store to an earlier configuration, viz. the configuration at the time the marker was created.

The state is a tuple of a store σ and some unspecified external component X . The external component is not recorded in markers, and thus not restored during backtracking.

The backtracking control flow is modelled using continuations. In contrast to denotational semantics, these continuations are encoded as first-order data structures that are interpreted by special-purpose relations.

7.3 Initial Dynamic Objects

The initial dynamic objects ME_{init} , VE_{init} , and s_{init} , and the contents of the set PrimVal are defined in Section ??.

The function $APPLY(\text{prim}, v^*, s) \Rightarrow (v'^*, s')/(\text{FAIL}, s')$ describes the effect of calling a primitive procedure. It is also defined in Section ??.

7.4 Inference Rules

States and Markers

$$\boxed{\text{marker } s \Rightarrow m}$$

$$\frac{s = (\sigma, _)}{\text{marker } s \Rightarrow \sigma} \quad (101)$$

$$\boxed{\text{restore}(m, s) \Rightarrow s'}$$

$$\frac{s' = (\sigma, X)}{\text{restore}(\sigma, (_, X)) \Rightarrow s'} \quad (102)$$

Patterns

$$\boxed{\text{match}(\text{pat}, v) \Rightarrow VE/\text{FAIL}}$$

$$\frac{}{\text{match}(_, v) \Rightarrow \{\}} \quad (103)$$

$$\frac{lit_1 = lit_2}{match(lit_1, lit_2) \Rightarrow \{\}} \quad (104)$$

$$\frac{lit_1 \neq lit_2}{match(lit_1, lit_2) \Rightarrow \text{FAIL}} \quad (105)$$

$$\frac{con_1 = con_2}{match(\langle _ \rangle con_1, (con_2, _)) \Rightarrow \{\}} \quad (106)$$

$$\frac{con_1 \neq con_2}{match(\langle _ \rangle con_1, (con_2, _)) \Rightarrow \text{FAIL}} \quad (107)$$

$$\frac{con_1 = con_2 \quad match^*(pat^{(k)}, v^*) \Rightarrow VE/\text{FAIL}}{match(\langle _ \rangle con_1(pat_1, \dots, pat_k), (con_2, v^*)) \Rightarrow VE/\text{FAIL}} \quad (108)$$

$$\frac{con_1 \neq con_2}{match(\langle _ \rangle con_1(pat_1, \dots, pat_k), (con_2, v^*)) \Rightarrow \text{FAIL}} \quad (109)$$

$$\frac{match^*(pat^{(k)}, v^*) \Rightarrow VE/\text{FAIL}}{match((pat_1, \dots, pat_k), v^*) \Rightarrow VE/\text{FAIL}} \quad (110)$$

$$\frac{match(pat, v) \Rightarrow VE}{match(var \text{ as } pat, v) \Rightarrow VE + \{var \mapsto v\}} \quad (111)$$

$$\frac{match(pat, v) \Rightarrow \text{FAIL}}{match(var \text{ as } pat, v) \Rightarrow \text{FAIL}} \quad (112)$$

$$\boxed{match^*(pat^*, v^*) \Rightarrow VE/\text{FAIL}}$$

$$\frac{}{match^*([], []) \Rightarrow \{\}} \quad (113)$$

$$\frac{match(pat, v) \Rightarrow \text{FAIL}}{match^*(pat :: pat^*, v :: v^*) \Rightarrow \text{FAIL}} \quad (114)$$

$$\frac{match(pat, v) \Rightarrow VE \quad match^*(pat^*, v^*) \Rightarrow \text{FAIL}}{match^*(pat :: pat^*, v :: v^*) \Rightarrow \text{FAIL}} \quad (115)$$

$$\frac{match(pat, v) \Rightarrow VE \quad match^*(pat^*, v^*) \Rightarrow VE'}{match^*(pat :: pat^*, v :: v^*) \Rightarrow VE + VE'} \quad (116)$$

Long Variables

$$\boxed{lookupLongVar(longvar, ME, VE) \Rightarrow v}$$

$$\frac{VE(var) = v}{lookupLongVar(var, ME, VE) \Rightarrow v} \quad (117)$$

$$\frac{ME(modid) = VE' \quad VE'(var) = v}{lookupLongVar(modid.var, ME, VE) \Rightarrow v} \quad (118)$$

Expressions

$$\boxed{eval(exp, ME, VE) \Rightarrow v}$$

$$\overline{eval(lit, ME, VE) \Rightarrow lit \text{ in Val}} \quad (119)$$

$$\overline{eval(\langle modid. \rangle con, ME, VE) \Rightarrow (con, [])} \quad (120)$$

$$\frac{lookupLongVar(longvar, ME, VE) \Rightarrow v}{eval(longvar, ME, VE) \Rightarrow v} \quad (121)$$

$$\frac{eval^*(exp^{(k)}, ME, VE) \Rightarrow v^{(k)}}{eval(\langle modid. \rangle con(exp_1, \dots, exp_k), ME, VE) \Rightarrow (con, v^{(k)})} \quad (122)$$

$$\frac{eval^*(exp^{(k)}, ME, VE) \Rightarrow v^{(k)}}{eval((exp_1, \dots, exp_k), ME, VE) \Rightarrow v^{(k)} \text{ in Val}} \quad (123)$$

$$\boxed{eval^*(exp^{(k)}, ME, VE) \Rightarrow v^{(k)}}$$

$$\frac{eval(exp_i, ME, VE) \Rightarrow v_i \ (1 \leq i \leq k)}{eval^*(exp^{(k)}, ME, VE) \Rightarrow v^{(k)}} \quad (124)$$

Recursive Values

$$\boxed{unfold_v(VE_{rec}, v) \Rightarrow v'}$$

$$\overline{unfold_v(VE_{rec}, (clause, ME, VE, -)) \Rightarrow (clause, ME, VE, VE_{rec})} \quad (125)$$

$$\frac{v \notin \text{Closure}}{unfold_v(VE_{rec}, v) \Rightarrow v} \quad (126)$$

Comment: ?? VE_{rec} is the environment in which this closure was bound. The effect of this step is to create a new closure in which an additional level of recursion is available.

Recursive Value Environments

$$\boxed{\text{unfold}_{\text{VE}} \text{VE} \Rightarrow \text{VE}'}$$

$$\frac{\begin{array}{c} \text{VE} = \{ \text{var}_1 \mapsto v_1, \dots, \text{var}_k \mapsto v_k \} \\ \text{unfold}_v(\text{VE}, v_i) \Rightarrow v'_i \ (1 \leq i \leq k) \end{array}}{\text{unfold}_{\text{VE}} \text{VE} \Rightarrow \{ \text{var}_1 \mapsto v'_1, \dots, \text{var}_k \mapsto v'_k \}} \quad (127)$$

Comment: This rule unfolds every closure bound in the environment, thus enabling each of them to recurse one step.

Failure Continuations

$$\boxed{\text{fail}(\text{fc}, s) \Rightarrow a}$$

$$\frac{\text{restore}(m, s) \Rightarrow s' \quad \text{invoke}(\text{clause}, v^*, \text{ME}, \text{VE}, \text{fc}, \text{pc}, s') \Rightarrow a}{\text{fail}(\text{orElse}(m, \text{clause}, v^*, \text{ME}, \text{VE}, \text{fc}, \text{pc}), s) \Rightarrow a} \quad (128)$$

$$\frac{\text{restore}(m, s) \Rightarrow s' \quad \text{proceed}(\text{gc}, \text{VE}, \text{fc}, s') \Rightarrow a}{\text{fail}(\text{ornot}(m, \text{gc}, \text{VE}, \text{fc}), s) \Rightarrow a} \quad (129)$$

$$\frac{}{\text{fail}(\text{orhalt}, s) \Rightarrow \text{No}} \quad (130)$$

Goal Continuations

$$\boxed{\text{proceed}(\text{gc}, \text{VE}, \text{fc}, s) \Rightarrow a}$$

$$\frac{\text{exec}(\text{goal}, \text{ME}, \text{VE}, \text{fc}, \text{gc}, s) \Rightarrow a}{\text{proceed}(\text{andthen}(\text{goal}, \text{ME}, \text{gc}), \text{VE}, \text{fc}, s) \Rightarrow a} \quad (131)$$

$$\frac{\text{fail}(\text{fc}', s) \Rightarrow a}{\text{proceed}(\text{andnot}(\text{fc}'), \text{VE}, \text{fc}, s) \Rightarrow a} \quad (132)$$

$$\frac{\text{eval}^*(\text{exp}^*, \text{ME}, \text{VE}) \Rightarrow v^* \quad \text{return}(\text{pc}, v^*, s) \Rightarrow a}{\text{proceed}(\text{andreturn}(\text{ME}, \text{exp}^*, \text{pc}), \text{VE}, \neg, s) \Rightarrow a} \quad (133)$$

$$\frac{\text{fail}(\text{fc}, s) \Rightarrow a}{\text{proceed}(\text{andreturn}(\text{ME}, \text{fail}, \text{pc}), \text{VE}, \neg, s) \Rightarrow a} \quad (134)$$

$$\frac{}{\text{proceed}(\text{andhalt}, \text{VE}, \text{fc}, s) \Rightarrow \text{Yes}} \quad (135)$$

Comment: ?? The failure continuation in effect at the time of the return is abandoned in favour of the one recorded in the procedure continuation pc . This restricts relations to be determinate.

Procedure Continuations

$$\boxed{\text{return}(\text{pc}, v^*, s) \Rightarrow a}$$

$$\frac{\text{match}^*(\text{pat}^*, v^*) \Rightarrow \text{VE}' \quad \text{proceed}(\text{gc}, \text{VE} + \text{VE}', \text{fc}, s) \Rightarrow a}{\text{return}(\text{retmatch}(\text{pat}^*, \text{VE}, \text{gc}, \text{fc}), v^*, s) \Rightarrow a} \quad (136)$$

$$\frac{match^*(pat^*, v^*) \Rightarrow \text{FAIL} \quad fail(fc, s) \Rightarrow a}{return(\text{retmatch}(pat^*, VE, gc, fc), v^*, s) \Rightarrow a} \quad (137)$$

Procedure Calls

$$\boxed{call(v, v^*, fc, pc, s) \Rightarrow a}$$

$$\frac{unfold_{VE} VE_{rec} \Rightarrow VE'_{rec} \quad invoke(\text{clause}, v^*, ME, VE + VE'_{rec}, fc, pc, s) \Rightarrow a}{call((\text{clause}, ME, VE, VE_{rec}), v^*, fc, pc, s) \Rightarrow a} \quad (138)$$

$$\frac{APPLY(\text{prim}, v^*, s) \Rightarrow (v'^*, s') \quad return(pc, v'^*, s') \Rightarrow a}{call(\text{prim}, v^*, fc, pc, s) \Rightarrow a} \quad (139)$$

$$\frac{APPLY(\text{prim}, v^*, s) \Rightarrow (\text{FAIL}, s') \quad fail(fc, s') \Rightarrow a}{call(\text{prim}, v^*, fc, pc, s) \Rightarrow a} \quad (140)$$

Goals

$$\boxed{exec(goal, ME, VE, fc, gc, s) \Rightarrow a}$$

$$\frac{\begin{array}{l} lookupLongVar(longvar, ME, VE) \Rightarrow v \\ eval^*(exp^*, ME, VE) \Rightarrow v'^* \quad pc = \text{retmatch}(pat^*, VE, gc, fc) \\ call(v, v'^*, fc, pc, s) \Rightarrow a \end{array}}{exec(longvar \text{ exp}^* => pat^*, ME, VE, fc, gc, s) \Rightarrow a} \quad (141)$$

$$\frac{\begin{array}{l} eval(exp, ME, VE) \Rightarrow v \quad VE(var) = v' \\ v = v' \quad proceed(gc, VE, fc, s) \Rightarrow a \end{array}}{exec(var = exp, ME, VE, fc, gc, s) \Rightarrow a} \quad (142)$$

$$\frac{eval(exp, ME, VE) \Rightarrow v \quad VE(var) = v' \quad v \neq v' \quad fail(fc, s) \Rightarrow a}{exec(var = exp, ME, VE, fc, gc, s) \Rightarrow a} \quad (143)$$

$$\frac{\begin{array}{l} eval(exp, ME, VE) \Rightarrow v \quad match(pat, v) \Rightarrow VE' \\ proceed(gc, VE + VE', fc, s) \Rightarrow a \end{array}}{exec(\text{let } pat = exp, ME, VE, fc, gc, s) \Rightarrow a} \quad (144)$$

$$\frac{eval(exp, ME, VE) \Rightarrow v \quad match(pat, v) \Rightarrow \text{FAIL} \quad fail(fc, s) \Rightarrow a}{exec(\text{let } pat = exp, ME, VE, fc, gc, s) \Rightarrow a} \quad (145)$$

$$\frac{\begin{array}{l} marker \ s \Rightarrow m \quad fc' = \text{ornot}(m, gc, VE, fc) \quad gc' = \text{andnot}(fc) \\ exec(goal, ME, VE, fc', gc', s) \Rightarrow a \end{array}}{exec(\text{not } goal, ME, VE, fc, gc, s) \Rightarrow a} \quad (146)$$

$$\frac{gc' = \mathbf{andthen}(goal_2, ME, gc) \quad exec(goal_1, ME, VE, fc, gc', s) \Rightarrow a}{exec(goal_1 \ \& \ goal_2, ME, VE, fc, gc, s) \Rightarrow a} \quad (147)$$

Comment: ?? Two values are equal if their representations are structurally identical. Since references are represented by their locations in the store, two references are equal if and only if they are in fact the same. The type discipline (in particular, the fact that relations do not admit equality) ensures that equality is only ever applied to literals, nullary constructors, locations, and values built out of such by tuple formation and constructor application.

$$\boxed{exec'(\langle goal \rangle, ME, VE, fc, gc, s) \Rightarrow a}$$

$$\frac{exec(goal, ME, VE, fc, gc, s) \Rightarrow a}{exec'(goal, ME, VE, fc, gc, s) \Rightarrow a} \quad (148)$$

$$\frac{proceed(gc, VE, fc, s) \Rightarrow a}{exec'(\varepsilon, ME, VE, fc, gc, s) \Rightarrow a} \quad (149)$$

Clauses

$$\boxed{invoke(clause, v^*, ME, VE, fc, pc, s) \Rightarrow a}$$

$$\frac{match^*(pat^*, v^*) \Rightarrow VE' \quad gc = \mathbf{andreturn}(ME, result, pc) \quad exec'(\langle goal \rangle, ME, VE + VE', fc, gc, s) \Rightarrow a}{invoke(\mathbf{rule} \ \langle goal \rangle \ \mathbf{--} \ _ \ pat^* \Rightarrow result, v^*, ME, VE, fc, pc, s) \Rightarrow a} \quad (150)$$

$$\frac{match^*(pat^*, v^*) \Rightarrow \mathbf{FAIL} \quad fail(fc, s) \Rightarrow a}{invoke(\mathbf{rule} \ \langle goal \rangle \ \mathbf{--} \ _ \ pat^* \Rightarrow result, v^*, ME, VE, fc, pc, s) \Rightarrow a} \quad (151)$$

$$\frac{marker \ s \Rightarrow m \quad fc' = \mathbf{orelse}(m, clause_2, v^*, ME, VE, fc, pc) \quad invoke(clause_1, v^*, ME, VE, fc', pc, s) \Rightarrow a}{invoke(clause_1 \ clause_2, v^*, ME, VE, fc, pc, s) \Rightarrow a} \quad (152)$$

Relation Bindings

$$\boxed{evalRel(ME, VE, relbind) \Rightarrow VE'}$$

$$\frac{VE' = \{var \mapsto (clause, ME, VE, \{\})\}}{evalRel(ME, VE, var \ \langle : \ ty \rangle = clause) \Rightarrow VE'} \quad (153)$$

$$\frac{evalRel(ME, VE, relbind_1) \Rightarrow VE_1 \quad evalRel(ME, VE, relbind_2) \Rightarrow VE_2}{evalRel(ME, VE, relbind_1 \ \mathbf{and} \ relbind_2) \Rightarrow VE_1 + VE_2} \quad (154)$$

Declarations

$$\boxed{evalDec(ME, VE, dec) \Rightarrow VE'}$$

$$\frac{eval(exp, ME, VE) \Rightarrow v}{evalDec(ME, VE, \mathbf{val} \text{ } var = exp) \Rightarrow VE + \{var \mapsto v\}} \quad (155)$$

$$\frac{evalRel(ME, VE, relbind) \Rightarrow VE' \quad unfold_{VE} VE' \Rightarrow VE''}{evalDec(ME, VE, \mathbf{relation} \text{ } relbind) \Rightarrow VE + VE''} \quad (156)$$

$$\frac{evalDec(ME, VE, dec_1) \Rightarrow VE_1 \quad evalDec(ME, VE_1, dec_2) \Rightarrow VE_2}{evalDec(ME, VE, dec_1 \text{ } dec_2) \Rightarrow VE_2} \quad (157)$$

$$\frac{dec = \mathbf{with} \text{ } _ \vee dec = \mathbf{type} \text{ } _ \vee dec = \mathbf{datatype} \text{ } _}{evalDec(ME, VE, dec) \Rightarrow VE} \quad (158)$$

Module Sequences

$$\boxed{load(ME, modseq) \Rightarrow ME'}$$

$$\frac{evalDec(ME, VE_{init}, dec) \Rightarrow VE}{load(ME, \mathbf{module} \text{ } modid : _ \mathbf{end} \text{ } dec) \Rightarrow ME + \{modid \mapsto VE\}} \quad (159)$$

$$\frac{load(ME, modseq_1) \Rightarrow ME_1 \quad load(ME_1, modseq_2) \Rightarrow ME_2}{load(ME, modseq_1 \text{ } modseq_2) \Rightarrow ME_2} \quad (160)$$

Program Arguments

$$\boxed{cnvargv \text{ } scon^* \Rightarrow v}$$

$$\overline{cnvargv \text{ } [] \Rightarrow (\mathbf{nil}, [])} \quad (161)$$

$$\frac{v = (scon \text{ in Lit}) \text{ in Val} \quad cnvargv \text{ } scon^* \Rightarrow v'}{cnvargv \text{ } scon :: scon^* \Rightarrow (\mathbf{cons}, [v, v'])} \quad (162)$$

Programs

$$\boxed{run(modseq, scon^*) \Rightarrow a}$$

$$\frac{\begin{array}{l} load(ME_{init}, modseq) \Rightarrow ME \quad ME(\mathbf{Main}) = VE \quad VE(\mathbf{main}) = v \\ cnvargv \text{ } scon^* \Rightarrow v' \quad fc = \mathbf{orhalt} \\ pc = \mathbf{retmatch}([], \{\}, \mathbf{andhalt}, fc) \quad call(v, [v'], fc, pc, s_{init}) \Rightarrow a \end{array}}{run(modseq, scon^*) \Rightarrow a} \quad (163)$$

8 Initial Objects

This section defines the initial objects for the static and dynamic semantics. Although there is some overlap in naming (ME_{init} and VE_{init} occur in both parts), the static and dynamic semantics are completely separated.

8.1 Initial Static Objects

Figures ?? to ?? show the interface to the standard types, constructors, values, and relations.

The initial static objects are built in several steps. First let t_{lvar} be the type name (**RML**, **lvar**, **true**). The set MutTyName is $\{t_{\text{lvar}}\}$. Then assume that references to ME_{init} , TE_{init} , and VE_{init} in the inference rules for the static semantics are replaced by empty environments $\{\}$. Let TE and VE be the environments resulting from the elaboration of the **RML** interface. Now, $TE_{\text{init}} = TE$, $VE_{\text{init}} = VE$, and $ME_{\text{init}} = \{\mathbf{RML} \mapsto (TE_{\text{init}}, VE_{\text{init}})\}$.

Furthermore, let τ_{char} , τ_{int} , τ_{real} , and τ_{string} be the types corresponding to the **char**, **int**, **real**, and **string** type constructors in TE_{init} , and let t_{list} be the type name corresponding to the **list** type constructor in TE_{init} .

8.2 Initial Dynamic Objects

The set PrimVal is equal to the set of variable identifiers bound as relations in the standard **RML** interface. $VE_{\text{init}} = \{var \mapsto prim ; prim \in \text{PrimVal} \wedge var = prim\}$, $ME_{\text{init}} = \{\mathbf{RML} \mapsto VE_{\text{init}}\}$, and $s_{\text{init}} = (\{\}, X)$ for some unspecified external component X of the state.

8.2.1 Primitive Procedures

The function *APPLY* describes the effect of calling a primitive procedure. We let *true* denote the value of the **true** constructor, i.e. $(\mathbf{true}, [])$ (similarly for *false*), *NONE* denote the value of the **NONE** constructor, i.e. $(\mathbf{NONE}, [])$, and *SOME*(*v*) denote $(\mathbf{SOME}, [v])$.

- $APPLY(\mathbf{clock}, [], s) = ([r], s)$ where r is a real number containing the number of seconds since some arbitrary but fixed (for the current process) past time point. The precision of r is unspecified.
- $APPLY(\mathbf{print}, [str], (\sigma, X)) = ([], (\sigma, X'))$ where X has been modified into X' by emitting the string str to the standard output device.
- $APPLY(\mathbf{tick}, [], (\sigma, X)) = ([i], (\sigma, X'))$ where i is an integer generated from X , and X' is X where this fact has been recorded so that i is not generated again.
- $APPLY(\mathbf{lvar_new}, [], (\sigma, X)) = ([l], (\sigma + \{l \mapsto \text{unbound}\}, X))$, where $l \notin \text{Dom } \sigma$.

```

module RML:
  (* types *)
  eqtype char
  eqtype int
  eqtype real
  eqtype string
  eqtype 'a vector
  eqtype 'a lvar (* admits eq, even if 'a does not *)
  datatype bool      = false
                    | true
  datatype 'a list    = nil
                    | cons of 'a * 'a list
  datatype 'a option  = NONE
                    | SOME of 'a

  (* booleans *)
  relation bool_and:  (bool,bool) => bool
  relation bool_not:  bool => bool
  relation bool_or:   (bool,bool) => bool
  (* characters *)
  relation char_int:  char => int
  (* integers *)
  relation int_abs:   int => int
  relation int_add:   (int,int) => int
  relation int_char:  int => char
  relation int_div:   (int,int) => int
  relation int_eq:    (int,int) => bool
  relation int_ge:    (int,int) => bool
  relation int_gt:    (int,int) => bool
  relation int_le:    (int,int) => bool
  relation int_lt:    (int,int) => bool
  relation int_max:   (int,int) => int
  relation int_min:   (int,int) => int
  relation int_mod:   (int,int) => int
  relation int_mul:   (int,int) => int
  relation int_ne:    (int,int) => bool
  relation int_neg:   int => int
  relation int_real:  int => real
  relation int_string: int => string
  relation int_sub:   (int,int) => int

```

Figure 23: Interface of the standard RML module

```

(* reals *)
relation real_abs:      real => real
relation real_add:      (real,real) => real
relation real_atan:     real => real
relation real_cos:      real => real
relation real_div:      (real,real) => real
relation real_eq:       (real,real) => bool
relation real_exp:      real => real
relation real_floor:    real => real
relation real_ge:       (real,real) => bool
relation real_gt:       (real,real) => bool
relation real_int:      real => int
relation real_le:       (real,real) => bool
relation real_ln:       real => real
relation real_lt:       (real,real) => bool
relation real_max:      (real,real) => real
relation real_min:      (real,real) => real
relation real_mod:      (real,real) => real
relation real_mul:      (real,real) => real
relation real_ne:       (real,real) => bool
relation real_neg:      real => real
relation real_pow:      (real,real) => real
relation real_sin:      real => real
relation real_sqrt:     real => real
relation real_string:   real => string
relation real_sub:      (real,real) => real

(* strings *)
relation string_append: (string,string) => string
relation string_int:    string => int
relation string_length: string => int
relation string_list:   string => char list
relation string_nth:    (string,int) => char

(* vectors *)
relation vector_length: 'a vector => int
relation vector_list:   'a vector => 'a list
relation vector_nth:    ('a vector,int) => 'a

```

Figure 24: Interface of the standard RML module (contd.)


```

(* lists *)
relation list_append: ('a list, 'a list) => 'a list
relation list_delete: ('a list, int) => 'a list
relation list_length: 'a list => int
relation list_member: ('a, 'a list) => bool
relation list_nth: ('a list, int) => 'a
relation list_reverse: 'a list => 'a list
relation list_string: char list => string
relation list_vector: 'a list => 'a vector
(* logical variables *)
relation lvar_new: () => 'a lvar
relation lvar_get: 'a lvar => 'a option
relation lvar_set: ('a lvar, 'a) => ()
(* miscellaneous *)
relation clock: () => real
relation print: string => ()
relation tick: () => int
end

```

Figure 25: Interface of the standard RML module (contd.)

- $APPLY(\text{lvar_get}, [l], (\sigma, X)) = ([NONE], (\sigma, X))$, if $\sigma(l) = \text{unbound}$, and $([SOME(v)], (\sigma, X))$, if $\sigma(l)$ is the value v .
- $APPLY(\text{lvar_set}, [l, v], (\sigma, X)) = ([], (\sigma + \{l \mapsto v\}, X))$, if $\sigma(l) = \text{unbound}$, or $(FAIL, (\sigma, X))$ if $\sigma(l) \in \text{Val}$.

The following operations do not access the state. If an operation succeeds yielding a value v , then $APPLY$ returns $([v], s)$. If an operation fails, then $APPLY$ returns $(FAIL, s)$. Below, we abbreviate $APPLY(\text{prim}, [x], s)$ to $\text{prim}(x)$, and analogously for operations with more arguments.

- $\text{int_abs}(i)$ returns the absolute value of i .
- $\text{int_add}(i_1, i_2) = i_1 + i_2$
- $\text{int_div}(i_1, i_2)$ returns the integer quotient of i_1 and i_2 ; if $i_2 = 0$, the operation fails.
- $\text{int_eq}(i_1, i_2) = \text{true}$ if $i_1 = i_2$, *false* otherwise.
- $\text{int_ge}(i_1, i_2) = \text{true}$ if $i_1 \geq i_2$, *false* otherwise.
- $\text{int_gt}(i_1, i_2) = \text{true}$ if $i_1 > i_2$, *false* otherwise.
- $\text{int_le}(i_1, i_2) = \text{true}$ if $i_1 \leq i_2$, *false* otherwise.

- `int_lt(i_1, i_2)` = *true* if $i_1 < i_2$, *false* otherwise.
- `int_max(i_1, i_2)` = i_1 if $i_1 \geq i_2$, i_2 otherwise.
- `int_min(i_1, i_2)` = i_1 if $i_1 \leq i_2$, i_2 otherwise.
- `int_mod(i_1, i_2)` returns the integer remainder of i_1 and i_2 ; if $i_2 = 0$, the operation fails.
- `int_mul(i_1, i_2)` = $i_1 \times i_2$
- `int_ne(i_1, i_2)` = *true* if $i_1 \neq i_2$, *false* otherwise.
- `int_neg(i)` = $-i$
- `int_real(i)` = r where r is the real value equal to i .
- `int_string(i)` returns a textual representation of i , as a string.
- `int_sub(i_1, i_2)` = $i_1 - i_2$
- `real_abs(r)` returns the absolute value of r .
- `real_add(r_1, r_2)` = $r_1 + r_2$.
- `real_atan(r)` returns the arc tangent of r .
- `real_cos(r)` returns the cosine of r (measured in radians).
- `real_div(r_1, r_2)` = r_1/r_2 ; if $r_2 = 0$, the operation fails.
- `real_eq(r_1, r_2)` = *true* if $r_1 = r_2$, *false* otherwise.
- `real_exp(r)` returns e^r .
- `real_floor(r)` returns the largest integer (as a real value) not greater than r .
- `real_ge(r_1, r_2)` = *true* if $r_1 \geq r_2$, *false* otherwise.
- `real_gt(r_1, r_2)` = *true* if $r_1 > r_2$, *false* otherwise.
- `real_int(r)` discards the fractional part of r and returns the integral part as an integer.
- `real_le(r_1, r_2)` = *true* if $r_1 \leq r_2$, *false* otherwise.
- `real_ln(r)` returns the natural logarithm of r ; fails if $r \leq 0$.
- `real_lt(r_1, r_2)` = *true* if $r_1 < r_2$, *false* otherwise.
- `real_max(r_1, r_2)` = r_1 if $r_1 \geq r_2$, r_2 otherwise.

- `real_min`(r_1, r_2) = r_1 if $r_1 \leq r_2$, r_2 otherwise.
- `real_mod`(r_1, r_2) returns the remainder of r_1/r_2 . This is the value $r_1 - i \times r_2$, for some integer i such that the result has the same sign as r_1 and magnitude less than the magnitude of r_2 . If $r_2 = 0$, the operation fails. (This corresponds to ANSI-C's `fmod` function.)
- `real_mul`(r_1, r_2) = $r_1 \times r_2$.
- `real_ne`(r_1, r_2) = *true* if $r_1 \neq r_2$, *false* otherwise.
- `real_neg`(r) = $-r$.
- `real_pow`(r_1, r_2) = $r_1^{r_2}$. This is defined when $r_1 > 0$, or $r_1 < 0$ and r_2 is integral, or when $r_1 = 0$ and $r_2 \geq 0$; in other cases, the operation fails. 0^0 is defined as 1.
- `real_sin`(r) returns the sine of r (measured in radians).
- `real_sqrt`(r) = \sqrt{r} ; if $r < 0$, the operation fails.
- `real_string`(r) returns a textual representation of r , as a string.
- `real_sub`(r_1, r_2) = $r_1 - r_2$.
- `string_int`(str) = i if the string has the lexical structure of an integer constant (as defined by the `Icon` token class) and i is the value associated with that constant. Otherwise the operation fails.

8.2.2 Implementation Dependent Parameters

Implicit in the description of the primitive procedures is that many may fail if their results cannot be represented by the implementation. We distinguish between four levels of conformance to this specification:

- Level 0 implementations have fixed precision integers and reals. Overflow and underflow conditions may or may not be reported as failures; instead, approximate values may be returned. Also, approximations are allowed for reals, e.g. by using IEEE floating-point.
- Level 1 implementations extend Level 0 implementations by detecting all overflow and underflow conditions, reporting them as failures.
- Level 2 implementations have infinite-precision integers, for which no overflow conditions are possible except due to memory exhaustion. Reals behave as in Level 1 implementations.
- Level 3 implementations have infinite precision integers and reals. No overflow or underflow conditions are possible except due to memory exhaustion.

An implementation *must* document its conformance level. It is recommended that IEEE 64-bit floating-point arithmetic be used for reals, and that at least 31 bits of precision be available for integers.

An implementation is *weakly* conforming if it implements only a subset of the standard types and operations described here. The implemented subset must conform to this document. An implementation is also free to *add* components to the standard interface.

8.2.3 Derived Dynamic Objects

The behaviour of some standard relations can be defined in RML itself; they include the boolean, character, list, and vector operations, and most string operations. Their definitions are shown in Figures ?? to ??. An implementation is expected to supply equivalent, but usually more efficient, implementations of some of these relations. In particular, although the vector and string types can be defined in terms of lists, the `vector_length`, `vector_nth`, `string_length` and `string_nth` relations are intended to execute in constant time.

```

relation bool_and =
  axiom bool_and(true, true) => true
  axiom bool_and(true, false) => false
  axiom bool_and(false, true) => false
  axiom bool_and(false, false) => false
end

relation bool_or =
  axiom bool_or(false, false) => false
  axiom bool_or(false, true) => true
  axiom bool_or(true, false) => true
  axiom bool_or(true, true) => true
end

relation bool_not =
  axiom bool_not false => true
  axiom bool_not true => false
end

datatype char = CHR of int (* [0,255] *)

relation char_int =
  axiom char_int(CHR i) => i
end

relation int_char =
  rule  int_ge(i,0) => true & int_le(i,255) => true
      -----
      int_char i => CHR i
end

```

Figure 26: Derived types and relations

```

relation list_append =
  axiom list_append([], y) => y

  rule  list_append(y, z) => yz
  -----
  list_append(x::y, z) => x::yz
end

relation list_reverse =
  axiom list_reverse [] => []

  rule  list_reverse y => revy &
  list_append(revy, [x]) => z
  -----
  list_reverse (x::y) => z
end

relation list_length =
  axiom list_length [] => 0

  rule  list_length xs => n & int_add(1, n) => n'
  -----
  list_length (_::xs) => n'
end

relation list_member =
  axiom list_member(_, []) => false

  rule  x = y
  -----
  list_member(x, y::ys) => true

  rule  not x = y & list_member(x, ys) => z
  -----
  list_member(x, y::ys) => z
end

```

Figure 27: Derived types and relations (contd.)

```

relation list_nth =
  axiom list_nth(x::_, 0) => x

  rule  int_sub(n, 1) => n' & list_nth(xs, n') => x
        -----
        list_nth(_::xs, n) => x
end

relation list_delete =
  axiom list_delete(_::xs, 0) => xs

  rule  int_sub(n, 1) => n' & list_delete(xs, n') => xs'
        -----
        list_delete(x::xs, n) => x::xs'
end

datatype 'a vector = VEC of 'a list

relation list_vector =
  axiom list_vector l => VEC l
end

relation vector_list =
  axiom vector_list(VEC l) => l
end

relation vector_length =
  rule  list_length l => n
        -----
        vector_length(VEC l) => n
end

relation vector_nth =
  rule  list_nth(l, n) => x
        -----
        vector_nth(VEC l, n) => x
end

```

Figure 28: Derived types and relations (contd.)

```

datatype string = STR of char list

relation list_string =
  axiom list_string cs => STR cs
end

relation string_list =
  axiom string_list(STR cs) => cs
end

relation string_length =
  rule  list_length cs => n
      -----
      string_length(STR cs) => n
end

relation string_nth =
  rule  list_nth(cs, n) => c
      -----
      string_nth(STR cs, n) => c
end

relation string_append =
  rule  list_append(cs1, cs2) => cs3
      -----
      string_append(STR cs1, STR cs2) => STR cs3
end

```

Figure 29: Derived types and relations (contd.)