The Definition of Extended ML

Stefan Kahrs* Donald Sannella* Andrzej Tarlecki†

— Version 1 —

Abstract

This document formally defines the syntax and semantics of the Extended ML language. It is based directly on the published semantics of Standard ML in an attempt to ensure compatibility between the two languages.

*LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland.

†Institute of Informatics, Warsaw University, and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

Contents

1 Introduction **1**
1.1 Behavioural equivalence . . . 3
1.2 Metalanguage . . . 3

2 Syntax of the Core **8**
2.1 Reserved Words . . . 8
2.2 Special constants . . . 8
2.3 Comments . . . 9
2.4 Identifiers . . . 9
2.5 Lexical analysis . . . 10
2.6 Infixed operators . . . 10
2.7 Derived Forms . . . 11
2.8 Grammar . . . 11
2.9 Syntactic Restrictions . . . 15

3 Syntax of Modules **16**
3.1 Reserved Words . . . 16
3.2 Identifiers . . . 16
3.3 Infixed operators . . . 16
3.4 Grammar for Modules . . . 16
3.5 Syntactic Restrictions . . . 16
3.6 Closure Restrictions . . . 18

4 Static Semantics for the Core **22**
4.1 Simple Objects . . . 22
4.2 Compound Objects . . . 23
4.3 Projection, Injection and Modification . . . 23
4.4 Types and Type functions . . . 25
4.5 Type Schemes . . . 26
Traces and Trace Schemes . . . 26
4.6 Scope of Explicit Type Variables . . . 26
4.7 Non-expansive Expressions . . . 27
4.8 Closure . . . 27
4.9 Type Structures and Type Environments . . . 27
4.10 Inference Rules . . . 28
4.11 Further Restrictions . . . 37

5 Static Semantics for Modules **38**
5.1 Semantic Objects . . . 38
5.2 Consistency . . . 39
5.3 Well-formedness . . . 39
5.4 Cycle-freedom . . . 40

5.5 Admissibility . . . 40
5.6 Type Realisation . . . 40
5.7 Realisation . . . 40
Traces . . . 41
Stripping Axioms . . . 41
5.8 Type Explication . . . 42
5.9 Signature Instantiation . . . 42
5.10 Functor Signature Instantiation . . . 42
5.11 Enrichment . . . 42
5.12 Signature Matching . . . 43
5.13 Principal Signatures . . . 43
5.14 Inference Rules . . . 45

6 Dynamic Semantics for the Core 55
6.1 Reduced Syntax . . . 55
6.2 Simple Objects . . . 55
6.3 Compound Objects . . . 56
6.4 Basic Values . . . 57
6.5 Basic Exceptions . . . 57
6.6 Closures . . . 58
States and Flags . . . 58
Pure Values . . . 59
6.7 Inference Rules . . . 63

7 Dynamic Semantics for Modules 73
7.1 Reduced Syntax . . . 73
7.2 Semantic Objects . . . 73
7.3 Inference Rules . . . 76

8 Verification Semantics for the Core 82
8.1 Semantic Objects . . . 82
8.2 Question Mark Interpretation . . . 82
8.3 State . . . 84
8.4 Value Access . . . 84
8.5 Type Interpretation . . . 85
8.6 Projections to Dynamic and Static Semantics . . . 86
8.7 Relationship to Dynamic Semantics . . . 86
8.8 Sentences of Static and Dynamic Semantics . . . 87
8.9 Inference Rules . . . 87

9 Verification Semantics for Modules 102
9.1 Compound Objects . . . 102
9.2 Generalised Axioms . . . 103
9.3 Combining Interfaces . . . 104

9.4 Extracting Objects of the Static Semantics . . . 105
9.5 Sets . . . 105
9.6 Inference Rules . . . 105

10 Programs **115**

A Appendix: Full Grammar **117**
A.1 Syntactic Restrictions . . . 124

B Appendix: Derived Forms **125**
B.1 Semantic Objects . . . 125
B.2 Identifiers . . . 126
B.3 Enrichment . . . 126
B.4 Inference Rules . . . 126

C Appendix: The Initial Basis **143**
C.1 The Initial State . . . 143
C.2 The Initial Context for Derived Forms . . . 143
C.3 The Initial Verification Basis . . . 145
C.4 Predefined Functions . . . 147

References **149**

Preface

Extended ML (EML) is a framework for the formal development of programs in the Standard ML (SML) programming language from high-level specifications of their required behaviour. The Extended ML language is a "wide-spectrum" language which encompasses both specifications and executable programs in a single unified framework. This allows all stages in the development of a program to be expressed in the Extended ML language, from the initial high-level specification to the final code itself and including intermediate stages in which specification and code are intermingled.

The Extended ML language is an extension of a large subset of Standard ML. This subset excludes references, assignment, input/output and imperative polymorphism, requires structure declarations and functor declarations to include explicit signatures, and restricts structures and functors to behave as abstractions[1] and parameterised abstractions respectively. Thus, Extended ML can only be used to specify/develop programs written in this subset of Standard ML. The Extended ML language extends this subset by permitting axioms in module interfaces (for specifying required properties of module components) and in place of code in module bodies (for describing functions in a non-algorithmic way prior to their implementation as Standard ML code).

The principles behind the design of the Extended ML language and development framework, details of its theoretical underpinnings and examples of its use may be found in [ST85], [ST86], [ST89], [San90] and [ST91]. The interested reader should consult these for background information. This document is a formal definition of the syntax and semantics of the Extended ML language; the other components of the Extended ML framework are disregarded here.

In order for Extended ML to serve its purpose as a framework for specifying and formally developing Standard ML programs, it is essential that the definition of the Extended ML language should appropriately "match" the published definition of Standard ML [MTH90]. Given the size of language definitions, such a match is practically impossible to achieve (let alone demonstrate in any convincing way) by *post hoc* comparison of two independent definitions. For this reason, the definition of Extended ML is based directly on the relevant parts of [MTH90], amended to correct errors and infelicities as described in Appendix D of [MT91] and as suggested in [Kah93] and [Kah94]. In order to make the relationship with Standard ML manifest, the structure of this document is as close as possible to that of [MTH90]. In places where the two languages are identical, the text of [MTH90] (with the indicated amendments) is used without change. For the most part, even the rule numbers and section numbers used in [MTH90] have been retained here. In detail:

[1] The term "abstraction" is taken from [MacQ86], the original description of the modules facilities of Standard ML. The idea is that only the information that is explicitly recorded in the signature(s) of a structure/functor is available to its clients.

- Sections 2–7 of this document correspond directly to Sections 2–7 of [MTH90]. Although there are a few more rules here than in [MTH90], corresponding rules appear with the same numbers for ease of comparison.
- Sections 8 and 9 are completely new.
- Section 10 here corresponds to Section 8 in [MTH90].
- Appendix A here corresponds directly to Appendix B there.
- Appendix B here is a reformulation of Appendix A there, which also takes care of identifier status (Appendix B of [MT91]) and infix directives.
- Appendix C here corresponds to Appendices C and D there.

The intention is that a "proof" that Extended ML is compatible with Standard ML, if such a thing could ever be constructed, would be based in large part on a simple textual comparison of the two definitions.

Because of the intimate relationship between [MTH90] and this document, familiarity with the former (for which study of [MT91] is strongly recommended!) is almost a prerequisite to achieving a deep understanding of the latter. The length and necessary formality of a definition such as this one makes it rather difficult to penetrate. For this reason an informal overview of the definition, which explains most of the main issues involved and justifies some of the choices taken, is provided in [KST94].

Acknowledgements

Thanks to Fabio da Silva for collaboration on an early version of the syntax and the static and dynamic semantics. Thanks to Thorsten Altenkirch, Mike Fourman, Robert Harley, Martin Hofmann, Ed Kazmierczak, Robin Milner and Mads Tofte for helpful discussions, comments and criticisms.

This document contains most of the text of *The Definition of Standard ML* by Robin Milner, Mads Tofte and Robert Harper (MIT Press, 1990), which is reproduced by kind permission of the authors and MIT Press.

The research reported here was partly supported by SERC grants GR/E78463 (SK, DS, AT), GR/H73103 (SK), GR/H76739 (AT), GR/J07303 (SK, DS) and GR/J07693 (DS), a SERC Advanced Fellowship (DS), the COMPASS Basic Research working group (DS, AT) and KBN grant 2 P301 007 04 (AT).

1 Introduction

This document formally defines the Extended ML language. As explained in the Preface, its structure closely follows that of the definition of Standard ML [MTH90]. Thus, apart from the usual separation between the definition of syntax and semantics, the semantics is divided into several parts:

- The *static semantics* deals with types (defining when a phrase *elaborates* to a type or an assembly containing types). It checks that phrases are well-typed, that they do not make reference to unbound identifiers, etc. We claim that the relations defined here are decidable, which means that mechanical type inference is possible.

- The *dynamic semantics* deals with values (defining when a phrase *evaluates* to a value). In Extended ML, phrases may specify values without defining them in an executable fashion. The result of evaluation in Extended ML is the same as in Standard ML, provided that such "undefined" values are not used in computing the result; otherwise a special exception is raised. Axioms in signatures and in structure/functor bodies are treated as formal comments by the dynamic semantics. We claim that the relations defined here are semi-decidable, which means that evaluation is implementable although (of course) it may fail to terminate.

- The *verification semantics* deals with the constraints imposed by axioms (defining when a phrase *verificates*[2] to a value or set of values). This includes checking that each structure and functor satisfies the axioms in its interface signature(s). Since axioms are not present in Standard ML, there is nothing in [MTH90] corresponding to the verification semantics. We claim that some of the relations defined here are not semi-decidable, which means that no complete proof system can exist for Extended ML.

Both syntax and semantics are further subdivided by treating the *Core* and *Modules* separately. The definition of syntax is also divided into the definition of the *Bare language*, which can be viewed as abstract syntax, and the definition of the *Full language* by (computable) translation of *derived forms* into the Bare language. The *initial basis* gives names and meanings to all the predefined identifiers in Extended ML. Finally, the semantics of "*programs*" completes the definition of the Extended ML language by combining the other parts of the definition. Figure 1 shows where all these parts appear in this document, and indicates the direct dependencies between the parts ($A \longrightarrow B$ means that A directly depends on B, i.e. A explicitly "calls" B). There are further indirect dependencies which must be kept in mind when reading the definition. The dynamic semantics depends on the static semantics, since evaluation of phrases is only guaranteed to be well-defined for phrases that elaborate. The verification semantics depends on

[2] A more obvious term is "verify", but this carries connotations we would like to avoid.

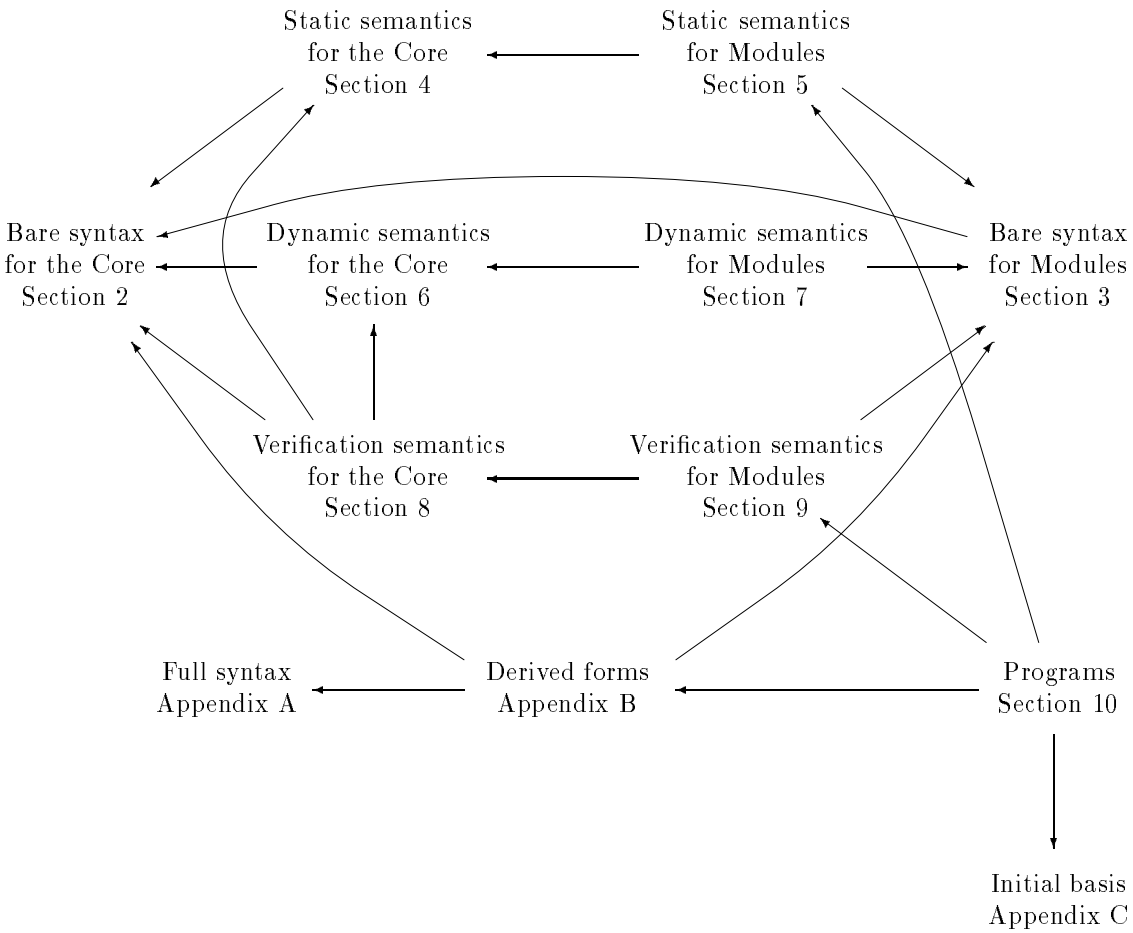

Figure 1: Parts of the definition of Extended ML

the static semantics in exactly the same fashion, with the difference that information is explicitly passed from the static semantics to the verification semantics via the program semantics. The initial basis depends on the dynamic semantics, the verification semantics and the derived forms in the sense that it contains semantic objects of classes defined in these sections. Since the metalanguage used for presenting the semantics is non-standard, it is presented later in this section.

1.1 Behavioural equivalence

The verification semantics defines what it means for a structure body to *match* its interface signature. Roughly speaking, this requires any model of the structure body to satisfy the axioms in the signature. Following ideas concerning the use of axioms to specify encapsulated abstractions, it is possible to relax this requirement by allowing the axioms to be satisfied not "literally", but only "up to behavioural equivalence" with respect to an appropriate set of "observable types" [ST89]. Similar remarks apply to functor declarations.

The present definition requires literal satisfaction of axioms. We intend to eventually change this to permit satisfaction up to behavioural equivalence, but further study is required before this can be done. Unfortunately, the approach used in previous work on the foundations of formal development in Extended ML, via a definition of behavioural equivalence between models, will not achieve the desired effect because of our use of models incorporating a rather concrete representation of types and values. We believe that a small modification to the meaning of quantification (rules 211–214) and logical equality (rules 231–232) is all that will be required. Before making this change we hope to show that there is a satisfactory relationship between what this yields and the behavioural equivalence relation used for the foundations of formal development, following [BHW94].

1.2 Metalanguage

The semantics of Extended ML is presented in a style known as Natural Semantics [Kah88], or rather an extended version of it. The metalanguage for the presentation of the semantics has *rules* of the form

$$\frac{\upsilon_1 \quad \cdots \quad \upsilon_k}{\psi}$$

where the *conclusion* ψ is a *sentence* and each *premise* (or *hypothesis*) υ_i is either a sentence or a rule. In particular we allow for the use of higher-order rules (with rules as premises).

In the presentation of the rules, phrases within single angle brackets $\langle\ \rangle$ are called *first options*, and those within double angle brackets $\langle\langle\ \rangle\rangle$ are called *second options*. To reduce the number of rules, we have adopted the following convention:

> In each instance of a rule, the first options must be either all present or all absent; similarly the second options must be either all present or all absent.

Thus, a rule abbreviates four rules if both first and second options occur in its presentation.

Primitive sentences are defined in the various sections of the semantics and are mostly of the form $B \vdash P \Rightarrow M$ where B and M are (tuples of) semantic objects and P is a syntactic object. Such a primitive sentence can be read as "against background B, the phrase P may be given the meaning M". The schema $_ \vdash _ \Rightarrow _$ can be seen as a family of ternary predicate symbols, which are defined by the semantics. These predicate symbols are overloaded for various phrase classes, but the context always resolves this ambiguity: the predicate symbol in the conclusion of a rule is always introduced by a *rule section* header, e.g. (cf. Section 4.10)

Expressions $\boxed{C \vdash exp \Rightarrow \tau, U, \gamma}$

introduces a predicate symbol $\vdash_{exp}$ in the static semantics of the Core. The following rules (up to the next rule section, in this particular case rules 9 to 14) all use this predicate symbol in their conclusion. The premises on the other hand use the "closest-fitting" predicate symbol, which will always be uniquely determined. For instance, rule 9 on page 30 in the static semantics seems to have identical premise and conclusion, but $\vdash$ in the conclusion refers to $\vdash_{exp}$ and $\vdash$ in the premise to $\vdash_{atexp}$. The meaning of "closest-fitting" is formalised by ordering the phrase classes of the syntax (languages generated by non-terminals of the grammar) by set inclusion, giving us a partial order on the mentioned predicate symbols: $\vdash_{atexp} \leq \vdash_{exp}$, because each atomic expression is an expression as well.

We also allow several other forms of primitive sentences (e.g. $x \in A$) and combine primitive sentences using logical connectives and quantifiers to form non-primitive sentences.

Metavariables within sentences range over syntactic and semantic objects. The name of a metavariable indicates the class of objects it ranges over. The name of a metavariable for a syntactic object is closely linked to the name of the corresponding *phrase class*, the language generated by the non-terminal. For example, the metavariable *exp* ranges over syntactic objects of the phrase class Exp, see Section 2.8. Concerning metavariables for semantic objects, the corresponding convention is introduced in the definition of the semantic objects, e.g. $t \in$ TyName in Figure 11 on page 22 introduces t as a metavariable ranging over semantic objects from TyName. This convention extends to priming and subscripting of metavariables, e.g. t'_1 is also a metavariable ranging over TyName. Finally, we write v/p for a metavariable which ranges over the (disjoint) union of the semantic object classes over which v and p range; they are called *compound* metavariables.

We interpret rules by translating them into formulae of many-sorted higher-order logic and then interpreting these formulae intuitionistically. Therefore, as

a whole, the semantics can be understood as a specification in higher-order logic. Its meaning is the set of consequences of this specification (derivable sentences) in higher-order intuitionistic logic. This approach sidesteps problems with the usual interpretation of such rules as giving an inductive definition of the predicates $_ \vdash _ \Rightarrow _$, since our use of negated premises and higher-order rules renders this interpretation potentially meaningless. However, it appears that the hierarchical structure of the semantics and the particular way in which the offending constructs are used make it possible to show that such an "inductive" interpretation would be unproblematic and would coincide with the interpretation we formally use here.

For the sake of legibility, the rules do not contain explicit quantifiers on all variables, but we employ instead a number of principles for inserting these in the course of the translation into higher-order logic. Besides the usual binding constructs (quantifiers) in logic, we understand rules (and set comprehensions) as *guarding* variables. If a variable is guarded by a rule then this fact corresponds to an implicit universal quantifier, provided the variable does not occur unguarded outside the rule. Thus, in contrast to usual binding, guarding operates top-down rather than bottom-up. Unguarded variables are free variables in the usual sense, but guarded variables can also be free, provided they are unguarded in the context of the formula. For instance, if we take implication $\Longrightarrow$ as a guarding operator in first-order logic then the formula $(P(x) \Longrightarrow Q(x,y)) \Longrightarrow R(y)$ is shorthand for $\forall y.\, (\forall x.\, (P(x) \Longrightarrow Q(x,y)) \Longrightarrow R(y))$, i.e. the variable x is bound at the inner implication, because it is not unguarded in the context $_ \Longrightarrow R(y)$.

Let us make this precise. If υ is a rule or sentence and V a set of variables, then $[\![\upsilon]\!]_V$ is a pair (V', Υ), where V' is a set of variables and Υ a formula in higher-order logic; we say that the variables in V' occur *unguarded* in υ. The set V in $[\![\upsilon]\!]_V$ serves here as the context of υ; it should be understood as the set of variables occurring unguarded outside υ. Similarly, if t is a term and V a set of variables, then $[\![t]\!]_V$ is a pair (t', V') where t' is a term and V' a set of variables.

We translate a top-level rule ϕ (i.e. one that is not being used as a rule premise) to a formula Φ, where $[\![\phi]\!]_{V_0} = (\emptyset, \Phi)$ (rules have no unguarded variables, see below) and V_0 are names for (the components of) the initial basis, see Appendix C. The properties maintained by the definition of $[\![\upsilon]\!]_V$ are that for $[\![\upsilon]\!]_V = (V', \phi)$, the free variables of ϕ are contained in $V \cup V'$, and if $[\![\upsilon]\!]_W = (W', \phi')$ then $V' = W'$, i.e. the set of unguarded variables does not depend on the second argument.

$$[\![\tfrac{\upsilon_1 \cdots \upsilon_n}{\psi}]\!]_V \;=\; (\emptyset, \forall x_1. \cdots . \forall x_k. (\Upsilon_1 \wedge \cdots \wedge \Upsilon_n \Longrightarrow \Psi))$$

$$\text{where} \begin{cases} (V_i, \Upsilon_i) &= [\![\upsilon_i]\!]_W \\ (V', \Psi) &= [\![\psi]\!]_W \\ W &= V_1 \cup \cdots \cup V_n \cup V' \cup V \\ \{x_1, \ldots, x_k\} &= W \setminus V \end{cases}$$

The above definition uniquely determines $[\![\frac{\upsilon_1 \cdots \upsilon_n}{\psi}]\!]_V$ up to permutation of bound variables. The cyclic dependency present in the definition can be resolved by first computing the first component of the result for each υ_i, the set of unguarded

variables — this does not depend on W. The symbol $\Longrightarrow$ in the definition refers to logical implication; it differs from $\Rightarrow$ which is part of our notation for primitive sentences.

For any sentence ψ, we put $[\![\psi]\!]_V = (V', \Psi)$, where Ψ is defined to be the same as ψ except that quantification in set comprehension is made explicit, see below; V' are the unguarded variables in ψ (w.r.t. V), where quantification and set comprehension are the only connectives that may guard variables. For a quantified sentence like $\exists x.\psi$ the unguarded variables w.r.t. V are $V' \setminus \{x\}$, where V' are the unguarded variables of ψ w.r.t. V. The unguarded variables of a primitive sentence $p(A_1, \ldots, A_n)$ w.r.t. V are the union of the unguarded variables of the A_i (w.r.t. the same V); the same principle applies to terms $f(A_1, \ldots, A_n)$. Each variable occurs unguarded in itself: $[\![x]\!]_V = (\{x\}, x)$, regardless of whether x is in V or not.

In contrast to the SML semantics, we allow rules as premises of rules. Based on the above translation scheme, such higher-order rules abbreviate formulae containing nested quantifiers, very much in the spirit of Hannan's "Extended Natural Semantics" [Han93]. Satisfaction of such a premise requires the rule to be admissible. Higher-order rules can be used to express "principality" and other infinitary requirements, as in the following rule 57 of the static semantics for Modules. Principality was described in English in [MTH90].

$$\frac{C \text{ of } B \vdash dec \Rightarrow E, \gamma \qquad N = \text{names}\,\gamma \setminus N \text{ of } B \qquad \frac{C \text{ of } B \vdash dec \Rightarrow E', \gamma'}{(N)\gamma \succ \gamma'}}{B \vdash dec \Rightarrow E, \gamma}$$

Applying the translation $[\![\ ]\!]_{V_0}$ to this rule, we obtain the following formula:

$$\begin{array}{l}\forall B.\, \forall dec.\, \forall E.\, \forall \gamma.\, \forall N. \\ \quad (C \text{ of } B \vdash dec \Rightarrow E, \gamma \ \wedge \ N = \text{names}\,\gamma \setminus N \text{ of } B \ \wedge \\ \qquad \forall E'.\, \forall \gamma'.\, (C \text{ of } B \vdash dec \Rightarrow E', \gamma' \ \Longrightarrow \ (N)\gamma \succ \gamma') \\ \quad \Longrightarrow B \vdash dec \Rightarrow E, \gamma)\end{array}$$

In expressions (of the metalanguage) such as C of B, C is not a variable but the name of a component of B-like objects. In the metalanguage, "$C$of" is a function symbol, the corresponding projection; see Section 4.3.

The important thing to remember about the translation from the metalanguage into formulae is how variables are scoped. In the example, E' and γ' are scoped at the local rule because they do not occur unguarded in any of the other components of the top-level rule. On the other hand, B, γ and N have such unguarded occurrences and so no quantifier is introduced for them at the local rule. This scoping principle of the metalanguage corresponds very closely to the scoping principle of explicit type variables in the object language (EML), where value declarations guard type variable occurrences; see Section 4.6.

Writing higher-order rules requires care in one special case, as the given translation scheme does not produce the formulae one might intuitively expect. It is the situation when a metavariable occurs in the conclusion of a local rule. Here is an example taken from rule 297 of the verification semantics for Modules:

$$\frac{s, B, \gamma \vdash \mathit{strdec} \Rightarrow \mathcal{E} \qquad \dfrac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{S}.\ s_1, B \oplus E, \gamma' \vdash \mathit{strexp} \Rightarrow \mathcal{S}}}{\begin{array}{l} s, B, \gamma \cdot \gamma' \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow \\ \quad \{(s_2, S) \mid (s_1, E) \in \mathcal{E},\ s_1, B \oplus E, \gamma' \vdash \mathit{strexp} \Rightarrow \mathcal{S},\ (s_2, S) \in \mathcal{S}\} \end{array}}$$

The problematic metavariable is $\mathcal{S}$. The existential quantifier in the conclusion of the premise is necessary to give the rule the intended meaning "we do not care what $\mathcal{S}$ is"; without this the meaning would be "we can do this for any $\mathcal{S}$". The same problem does not arise with *premises* of local rules: $\forall x.(P(x) \Longrightarrow Q)$ is the same as $(\exists x.P(x)) \Longrightarrow Q$. The above pattern of a higher-order rule combined with an existential quantifier in the conclusion of a premise occurs quite frequently in the verification semantics.

In the verification semantics for Modules, we use another form, having similar scoping principles as rules have, for describing certain semantic objects: set comprehension. A *set comprehension* denotes a semantic object (a possibly infinite set) and is of the form $\{A \mid \psi\}$ where A denotes a semantic object and ψ is a sentence. It is translated as follows:

$$\begin{array}{rcl} [\![\{A \mid \psi\}]\!]_V & = & (\emptyset, \{y \mid \exists x_1.\cdots.\exists x_k.(y = A' \wedge \psi')\}) \\ & & \text{where} \left\{ \begin{array}{rcl} (V', \psi') & = & [\![\psi]\!]_W \\ (W', A') & = & [\![A]\!]_W \\ W & = & V' \cup W' \cup V \\ \{x_1, \ldots, x_k\} & = & W \setminus V \end{array} \right. \end{array}$$

where y is a fresh variable. Like rules, set comprehensions have no unguarded variables.

We can remove set comprehensions entirely from the translated formulae as follows: a set comprehension $\{y \mid \psi\}$ is replaced by a fresh variable Y and the sentence in which it occurs is supplied with an additional premise $\forall y.(y \in Y \iff \psi)$. As a second-order rewrite rule:

$$\phi[\{y \mid \psi\}/x] \quad \longrightarrow \quad \forall Y.(\forall y.(y \in Y \iff \psi) \Longrightarrow \phi[Y/x]).$$

As our underlying logic is intuitionistic, these "sets" should be interpreted in a topos, see for instance [Gol84]. The difference is subtle but it matters as we shall encounter "sets" the membership predicate of which is undecidable.

2 Syntax of the Core

2.1 Reserved Words

The *reserved words* of Extended ML can be divided into two groups, namely:

1. those that are necessary for presenting the grammar of the Core; and
2. those additional reserved words that are needed for presenting the grammar for Modules.

Below we list reserved words of the first group; the rest are listed in Section 3.1. Reserved words may not (except `=`) be used as identifiers.

```
abstype   and   andalso   as    case   do
datatype   else   end   eqtype   exception   exists
forall   fn   fun   handle   if   implies   in   infix
infixr   let   local   nonfix   of   op   open   orelse
proper   raise   raises   rec   terminates
then   type   val   with   withtype   while
(  )   [  ]   {  }   ,   :   ;   ...   _   |  =   =>
->   #   *)   ?   ==   =/=
```

2.2 Special constants

An *integer constant* is any non-empty sequence of digits, possibly preceded by a negation symbol (`~`). A *real constant* is an integer constant, possibly followed by a point (`.`) and one or more digits, possibly followed by an exponent symbol `E` and an integer constant; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: `0.7  3.32E5  3E~7` . Non-examples: `23  .3  4.E5  1E2.0 .`

We assume an underlying alphabet of 256 characters (numbered 0 to 255) such that the characters with numbers 0 to 127 coincide with the ASCII character set. A *string constant* is a sequence, between quotes ("), of zero or more printable characters (i.e., numbered 33–126), spaces or escape sequences. Each escape sequence starts with the escape character `\` , and stands for a character sequence. The escape sequences are:

`\n`	A single character interpreted by the system as end-of-line.
`\t`	Tab.
`\^`c	The control character c, where c may be any character with number 64–95. The number of `\^`c is 64 less than the number of c.
`\`*ddd*	The single character with number *ddd* (3 decimal digits denoting an integer in the interval $[0, 255]$).

Var	(value variables)	long
Con	(value constructors)	long
ExCon	(exception constructors)	long
TyVar	(type variables)	
TyCon	(type constructors)	long
Lab	(record labels)	
StrId	(structure identifiers)	long

Figure 2: Identifiers

\"	"
\\	\
\$f\cdots f$\	This sequence is ignored, where $f\cdots f$ stands for a sequence of one or more formatting characters.

The *formatting characters* are a subset of the non-printable characters including at least space, tab, newline, formfeed. The last form allows long strings to be written on more than one line, by writing `\` at the end of one line and at the start of the next.

We denote by SCon the class of *special constants*, i.e., the integer, real, and string constants; we shall use *scon* to range over SCon.

2.3 Comments

A *comment* is any character sequence within comment brackets `(* *)` in which comment brackets are properly nested, i.e. the rules for forming lexical items do not apply within a comment. An unmatched comment bracket should be detected and rejected by the compiler.

No space is allowed between the two characters which make up a comment bracket `(*` or `*)`. Even an unmatched `*)` should be detected by the compiler. Thus the expression `(op *)` is illegal. But `(op * )` is legal; so is `op*` . Furthermore `(op **)` is legal because of the longest match principle for lexical analysis (see Section 2.5).

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 2. We use *var*, *tyvar* to range over Var, TyVar etc. For each class X marked "long" there is a class longX of *long identifiers*; if x ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

longx	::=	x	identifier
		$strid_1 . \cdots . strid_n . x$	qualified identifier ($n \geq 1$)

The qualified identifiers constitute a link between the Core and the Modules. Throughout this document, the term "identifier", occurring without an adjective, refers to non-qualified identifiers only.

An identifier is either *alphanumeric*: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or *symbolic*: any non-empty sequence of the following *symbols*

```
!  %  &  $  #  +  -  /  :  <  =  >  ?  @  \  ~  `  ^  |  *
```

In either case, however, reserved words are excluded. This means that for example `#` and `|` are not identifiers, but `##` and `|=|` are identifiers. The only exception to this rule is that the symbol `=` , which is a reserved word, is also allowed as an identifier to stand for the equality predicate. The identifier `=` may not be bound by the user; this precludes any syntactic ambiguity.

A *type variable tyvar* may be any alphanumeric identifier starting with a prime; the subclass EtyVar of TyVar, the *equality* type variables, consists of those which start with two or more primes. We exclude[3] identifiers from TyVar that start with one or two primes followed by an underbar. The other six classes (Var, Con, ExCon, TyCon, Lab and StrId) are represented by identifiers not starting with a prime. However, `*` is excluded from TyCon, to avoid confusion with the derived form of tuple type (see Figure 22). The class Lab is extended to include the *numeric* labels `1` `2` `3` $\cdots$, i.e. any numeral not starting with `0`.

Identifiers in the classes Var, Con, ExCon, TyCon and StrId all belong to the syntactic class Id. Within syntactic phrases (of the Bare Language), these subclasses are considered to be disjoint: for example, each $var \in$ Var has the form $id^{\mathtt{V}}$, being an identifier $id \in$ Id labelled with its status information — see Appendix B. Within other semantic objects, the labelling information has no significance. It is used, however, to disambiguate various overloaded forms of environment application: if an environment E contains a structure environment SE and a variable environment VE, then $E(id^{\mathtt{S}})$ is $SE(id)$ and $E(id^{\mathtt{V}})$ is $VE(id)$, etc.

2.5 Lexical analysis

Each item of lexical analysis is either a reserved word, a numeric label, a special constant or a long identifier. Comments and formatting characters separate items and are otherwise ignored. An exception from this rule are formatting characters within string constants; see Section 2.2. At each stage the longest next item is taken.

2.6 Infixed operators

An identifier may be given *infix status* by the `infix` or `infixr` directive, which may occur as a declaration. These declaration are not treated here, but see Appendices A and B.

[3]The reason for this exclusion is compatibility with Standard ML. The type variables in question are the *imperative* type variables of Standard ML.

atomic expressions
expression rows
expressions
matches
match rules

declarations
value bindings
type bindings
datatype bindings
constructor bindings
exception bindings

atomic patterns
pattern rows
patterns

type expressions
type-expression rows

Figure 3: Core Phrase Classes

2.7 Derived Forms

There are many standard syntactic forms in ML whose meaning can be expressed in terms of a smaller number of syntactic forms, called the *Bare* language. These derived forms, and their equivalent forms in the Bare language, are given in Appendix B. The rest of this document defines the syntax and semantics of the Bare language, with the exception of Appendices A (the full syntax) and B. The program semantics (Section 10) links the *Full* language with the semantics of the Bare language.

2.8 Grammar

The phrase classes for the Core are shown in Figure 3. We use the variable *atexp* to range over AtExp, etc.

The grammatical rules for the Core are shown in Figures 4, 5 and 6.

The following conventions are adopted in presenting the grammatical rules, and in their interpretation:

- The brackets $\langle\ \rangle$ enclose optional phrases.

- For any syntax class X (over which x ranges) we define the syntax class Xseq (over which *xseq* ranges) as follows:

xseq	::=	x	(singleton sequence)
			(empty sequence)
		$(x_1, \cdots, x_n)$	(sequence, $n \geq 1$)

Note that the "$\cdots$" used here, meaning syntactic iteration, must not be confused with "..." which is a reserved word of the language. To range over all three alternatives for sequences in semantic rules we write $x_1 \cdots x_n$ (with $n \geq 0$), which suppresses the syntactic commas and parentheses. The ambiguity for $n = 1$ will be harmless whenever we use this notation.

- Alternative forms for each phrase class are in order of decreasing ; this resolves ambiguity in parsing, as explained in Appendix A.

- Longest match: Suppose $F_1 F_2$ is an alternative form of a phrase class. A natural number i is called a *split index* w.r.t. $F_1 F_2$ for a lexical sequence $L_1 \cdots L_k$ if $0 \leq i \leq k$ and $L_1 \cdots L_i$ reduces to F_1 and $L_{i+1} \cdots L_k$ reduces to F_2. If for a given lexical sequence $L = L_1 \cdots L_k$ there are different split indices w.r.t. $F_1 F_2$, then L reduces to $F_1 F_2$ by reducing $L_1 \cdots L_j$ to F_1, where j is either the maximal split index, or — iff the alternative form is labelled (R), indicating a right associative infix construct — the minimal split index.

- For any syntax class X (over which x ranges) we define the syntax class $\mathrm{X}^\bullet$ (over which $x^\bullet$ ranges) as the same as X, except that phrases of class $\mathrm{X}^\bullet$ may not contain the lexical item `?`.

Notice that there is a difference between question marks for values and types: question marks for types have to be named, using the second form of *typbind*, while question marks for values can be anonymous. But a declaration of the form `val` *pat* `=` `?` is possible, because one form of expression *exp* is a question mark.

As a consequence of the refined disambiguation principle for precedence (see Appendix A) sequential declarations are given higher precedence than empty declarations. This is different from SML, but the definition of SML [MTH90] does not fully explain how parsing is affected by precedence.

atexp	::=	*scon*	special constant
		longvar	value variable
		longcon	value constructor
		longexcon	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	
		`?`	undefined value
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application
		exp `:` *ty*	typed
		$exp_1^\bullet$ `==` $exp_2^\bullet$	comparison (R)
		`exists` $match^\bullet$	existential quantifier
		`forall` $match^\bullet$	universal quantifier
		$exp^\bullet$ `terminates`	convergence predicate
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 4: Grammar: Expressions and Matches

dec	::=	`val` *valbind*	value declaration
		`type` *typbind*	type declaration
		`eqtype` *typbind*	equality type declaration
		`datatype` *datbind*	datatype declaration
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration ($n \geq 1$)
		dec_1 ⟨;⟩ dec_2	sequential declaration
			empty declaration
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨`and` *typbind*⟩	type binding
		tyvarseq tycon `=` `?` ⟨`and` *typbind*⟩	question mark type binding
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	*con* ⟨`of` *ty*⟩ ⟨ `|` *conbind*⟩	
exbind	::=	*excon* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		excon `=` *longexcon* ⟨`and` *exbind*⟩	

Figure 5: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		var	variable
		longcon	constant
		longexcon	exception constant
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
		longcon atpat	value construction
		longexcon atpat	exception construction
		pat `:` *ty*	typed
		var⟨`:` *ty*⟩ `as` *pat*	layered
ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type
		tyseq longtycon	type construction
		ty `->` *ty*′	function type (R)
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 6: Grammar: Patterns and Type Expressions

2.9 Syntactic Restrictions

- No pattern may contain the same *var* twice. No expression row, pattern row or type row may bind the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value constructors within a *datbind*.
- In the left side *tyvarseq tycon* of any *typbind* or *datbind*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring within the right side must occur in *tyvarseq*.
- For each value binding *pat* `=` *exp* within `rec`, *exp* must be of the form `fn` *match*[4]. The derived form of function-value binding given in Appendix B, rule 373, necessarily obeys this restriction.

[4]The SML definition adds here "possibly constrained by one or more type expressions". This is a void extension, because an expression `fn` *pat* `=>` *exp* `:` *ty* parses as `fn` *pat* `=>` (*exp* `:` *ty*).

3 Syntax of Modules

For Modules there are further reserved words, identifier classes and derived forms. There are no further special constants; comments and lexical analysis are as for the Core. The derived forms for modules concern mainly functors and appear in Appendix B.

3.1 Reserved Words

In addition to the listed in Section 2.1, Extended ML reserves the following words, which are used in the grammar for Modules:

```
axiom    functor    include     sharing
sig    signature     struct    structure
```

They may not be used as identifiers.

3.2 Identifiers

The additional identifier classes for Modules are (signature identifiers) and (functor identifiers); they may be either alphanumeric – not starting with a prime – or symbolic. Henceforth, we consider all identifier classes to be disjoint.

3.3 Infixed operators

Fixity directives and their scope are treated in Appendix B.

3.4 Grammar for Modules

The phrase classes for Modules are shown in Figure 7. We use the variable *strexp* to range over StrExp, etc. The conventions adopted in presenting the grammatical rules for Modules are the same as for the Core. The grammatical rules are shown in Figures 8, 9 and 10. Note that functor bindings and undefined structure bindings are required to include explicit (output) signatures.

Specification expressions *specexp* occur in axiom descriptions, i.e. in axioms within signatures. This is the only construct of the language in which structures can be declared as local to expressions. This is useful when one wants to express a property that depends on a functor instantiation.

3.5 Syntactic Restrictions

- No binding *strbind*, *sigbind*, or *funbind* may bind the same identifier twice.
- No description *valdesc*, *typdesc*, *datdesc*, *exdesc* or *strdesc* may describe the same identifier twice; this applies also to value constructors within a *datdesc*.

	structure expressions
	structure-level declarations
	axioms
	axiomatic expressions
	structure bindings
SglStrBind	single structure bindings
	signature expressions
	(principal) signature expressions
	signature declarations
	signature bindings
	specifications
	value descriptions
	type descriptions
	datatype descriptions
	constructor descriptions
	exception descriptions
	axiom descriptions
	specification expressions
	structure descriptions
	sharing equations
	functor declarations
	functor bindings
	top-level declarations

Figure 7: Modules Phrase Classes

- In the *tyvarseq tycon* in any *typdesc* or *datdesc*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring on the right side of the *datdesc* must occur in *tyvarseq*.

- In a single structure binding of the form *strid* `=` *strexp*, *strexp* must be guarded. A structure expression is called *guarded* if it is of the form *longstrid*, *funid* `(` *strexp* `)` or `let` *strdec* `in` *strexp* `end`, provided (in the last case) that *strexp* is guarded.

The last restriction is for purely methodological reasons: we want each structure to come equipped with an explicit signature. The reason why we include a single structure binding of the form *strid* `=` *strexp* in the language at all is the need to provide a way of realising structure sharing specifications, see [KST94]. The semantic rules for structure bindings do not exploit the guarding requirement.

3.6 Closure Restrictions

The semantics presented in later sections requires no restriction on reference to non-local identifiers. For example, it allows a signature expression to refer to external signature identifiers and (via `sharing` or `open`) to external structure identifiers; it also allows a functor (or structure expression) to refer to external identifiers of any kind.

However, implementers who want to provide a simple facility for separate compilation may want to impose the following restrictions (ignoring references to identifiers bound in the initial basis B_0, which may occur anywhere):

1. In any signature binding *sigid* `=` *psigexp* , the only non-local references in *psigexp* are to signature identifiers.

2. In any functor binding *funid* `(` *strid* `:` *psigexp* `)` `:` *psigexp*$'$ `=` *strexp* , the only non-local references in *psigexp*, *psigexp*$'$ and *strexp* are to functor and signature identifiers, except that both *psigexp*$'$ and *strexp* may refer to *strid* and its components.

In the second case the final qualification allows, for example, sharing constraints to be specified between functor argument and result. (For a completely precise definition of these closure restrictions, see the comments to rules 66 (page 48) and 96 (page 53) in the static semantics of modules, Section 5.)

The significance of these restrictions is that they may ease separate compilation and verification; this may be seen as follows. If one takes a *module* to be a sequence of signature declarations and functor declarations satisfying the above restrictions then the elaboration of a module can be made to depend on the initial static basis alone (in particular, it will not rely on structures outside the module). Moreover, the elaboration of a module cannot create new free structure or type names, so consistency (as defined in Section 5.2, page 39) is automatically preserved across separately compiled modules. On the other hand, imposing these restrictions may force the programmer to write many more sharing equations than is needed if functors and signature expressions can refer to free structures.

strexp	::=	`struct` *strdec* `end`	generative
		longstrid	structure identifier
		funid `(` *strexp* `)`	functor application
		`let` *strdec* `in` *strexp* `end`	local declaration
strdec	::=	*dec*	declaration
		`axiom` *ax*	axiom
		`structure` *strbind*	structure
		`local` $strdec_1$ `in` $strdec_2$ `end`	local
		$strdec_1$ ⟨`;`⟩ $strdec_2$	sequential
			empty
ax	::=	*axexp* ⟨`and` *ax*⟩	axiom
axexp	::=	$exp^{\bullet}$	axiomatic expression
strbind	::=	*sglstrbind* ⟨`and` *strbind*⟩	structure binding
sglstrbind	::=	*strid* `:` *psigexp* `=` *strexp*	single structure binding
		strid `:` *psigexp* `=` `?`	undefined structure binding
		strid `=` *strexp*	unguarded structure binding
sigexp	::=	`sig` *spec* `end`	generative
		sigid	signature identifier
psigexp	::=	*sigexp*	principal signature
sigdec	::=	`signature` *sigbind*	single
		$sigdec_1$ ⟨`;`⟩ $sigdec_2$	sequential
			empty
sigbind	::=	*sigid* `=` *psigexp* ⟨`and` *sigbind*⟩	

Figure 8: Grammar: Structure and Signature Expressions

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc*	datatype
		`exception` *exdesc*	exception
		`axiom` *axdesc*	axiom
		`structure` *strdesc*	structure
		`sharing` *shareq*	sharing
		`local` $spec_1$ `in` $spec_2$ `end`	local
		`open` $longstrid_1 \cdots longstrid_n$	open ($n \geq 1$)
		`include` $sigid_1 \cdots sigid_n$	include ($n \geq 1$)
		$spec_1$ $\langle$`;`$\rangle$ $spec_2$	sequential
			empty
valdesc	::=	*var* : *ty* $\langle$`and` *valdesc*$\rangle$	
typdesc	::=	*tyvarseq tycon* $\langle$`and` *typdesc*$\rangle$	
datdesc	::=	*tyvarseq tycon* `=` *condesc* $\langle$`and` *datdesc*$\rangle$	
condesc	::=	*con* $\langle$`of` *ty*$\rangle$ $\langle$ `|` *condesc*$\rangle$	
exdesc	::=	*excon* $\langle$`of` *ty*$\rangle$ $\langle$`and` *exdesc*$\rangle$	
axdesc	::=	*specexp* $\langle$`and` *axdesc*$\rangle$	
specexp	::=	`let` *strdec* `in` *axexp* `end`	
strdesc	::=	*strid* : *sigexp* $\langle$`and` *strdesc*$\rangle$	
shareq	::=	$longstrid_1$ `=` $\cdots$ `=` $longstrid_n$	structure sharing ($n \geq 2$)
		`type` $longtycon_1$ `=` $\cdots$ `=` $longtycon_n$	type sharing ($n \geq 2$)
		$shareq_1$ `and` $shareq_2$	multiple

Figure 9: Grammar: Specifications

fundec	::=	`functor` *funbind*	single
		fundec$_1$ ⟨`;`⟩ *fundec*$_2$	sequence
			empty
funbind	::=	*funid* `(` *strid* `:` *psigexp* `)` `:` *psigexp*′	
		`=` *strexp* ⟨`and` *funbind*⟩	functor binding
		funid `(` *strid* `:` *psigexp* `)` `:` *psigexp*′	
		`=` `?` ⟨`and` *funbind*⟩	undefined functor binding
topdec	::=	*strdec*	structure-level declaration
		sigdec	signature declaration
		fundec	functor declaration

Note: No *topdec* may contain, as an initial segment, a shorter top-level declaration followed by a semicolon.

Figure 10: Grammar: Functors and Top-level Declarations

4 Static Semantics for the Core

Our first task in presenting the semantics – whether for Core or Modules, static or dynamic – is to define the objects concerned. In addition to the class of *syntactic* objects, which we have already defined, there are classes of so-called *semantic* objects used to describe the meaning of the syntactic objects. Some classes contain *simple* semantic objects; such objects are usually identifiers or names of some kind. Other classes contain *compound* semantic objects, such as types or environments, which are constructed from component objects.

4.1 Simple Objects

All semantic objects in the static semantics of the entire language are built from identifiers and two further kinds of simple objects: type constructor names and structure names. Type constructor names are the values taken by type constructors; we shall usually refer to them briefly as type names, but they are to be clearly distinguished from type variables and type constructors. Structure names play an active role only in the Modules semantics; they enter the Core semantics only because they appear in structure environments, which (in turn) are needed in the Core semantics only to determine the values of long identifiers. The simple object classes, and the variables ranging over them, are shown in Figure 11. We have included TyVar in the table to make visible the use of α in the semantics to range over TyVar.

α or *tyvar*	$\in$	TyVar	type variables
t	$\in$	TyName	type names
m	$\in$	StrName	structure names

Figure 11: Simple Semantic Objects

The sets TyName and StrName are arbitrary infinite sets, except that TyName is a superset of T_0 (type names of the initial basis), see Section C.3.

Each $\alpha \in$ TyVar possesses a boolean *equality attribute*, which determines whether or not it *admits equality*, i.e. whether it is a member of EtyVar (defined on page 10). There is a distinguished type variable `num`; it is in EtyVar, but it has no syntactic representation in EML.

Each $t \in$ TyName has an arity $k \geq 0$, and also possesses an equality attribute. We denote the class of type names with arity k by $\text{TyName}^{(k)}$.

With each special constant *scon* we associate a type name type(*scon*) which is either `int`, `real` or `string` as indicated by Section 2.2.

4.2 Compound Objects

When A and B are sets $\mathrm{Fin}\, A$ denotes the set of finite subsets of A, and $A \xrightarrow{\mathrm{fin}} B$ 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 $\mathrm{Dom}\, f$ and $\mathrm{Ran}\, f$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$. We shall use the form $\{x \mapsto e \ ; \ \phi\}$ — a form of set comprehension — to stand for the finite map f whose domain is the set of values x which satisfy the condition ϕ, and whose value on this domain is given by $f(x) = e$. This notation slightly differs from those set comprehensions denoting sets (see page 7), for which we use a $\mid$ to separate the condition.

When f and g are finite maps the map $f + g$, called f *modified* by g, is the finite map with domain $\mathrm{Dom}\, f \cup \mathrm{Dom}\, g$ and values

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

For any semantic object class A, we define $\mathrm{Tree}(A)$ to be the finite binary trees of elements taken from A, i.e. $\mathrm{Tree}(A)$ is the smallest solution of the domain equation $\mathrm{Tree}(A) = \{\epsilon\} \uplus A \uplus \{x \cdot y \mid x, y \in \mathrm{Tree}(A)\}$. We take $\cdot$ to be left-associative, i.e. $x \cdot y \cdot z$ stands for $(x \cdot y) \cdot z$.

The compound objects for the static semantics of the Core Language are shown in Figure 12. We take $\uplus$ to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint.

Note that Λ and $\forall$ bind type variables. For any semantic or syntactic object A, $\mathrm{tynames}\, A$ and $\mathrm{tyvars}\, A$ denote respectively the set of type names and the set of type variables occurring free in A.

4.3 Projection, Injection and Modification

Projection: We often need to select components of tuples – for example, the variable-environment component of a context. In such cases we rely on variable names to indicate which component is selected. For instance "VE of E" means "the variable-environment component of E" and "m of S" means "the structure name of S".

Moreover, when a tuple contains a finite map we shall "apply" the tuple to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance $C(tycon)$ means $(TE \text{ of } C)tycon$.

A particular case needs mention: $C(con)$ is taken to stand for $(VE \text{ of } C)con$. The type scheme of a value constructor is held in VE as well as in TE (where it will be recorded within a CE). Thus the re-binding of a constructor is given proper effect by accessing it in VE, rather than in TE.

Finally, environments may be applied to long identifiers. For instance if $longcon = strid_1 . \cdots . strid_k . con$ then $E(longcon)$ means

$$(VE \text{ of } (SE \text{ of } \cdots (SE \text{ of } (SE \text{ of } E)strid_1)strid_2 \cdots)strid_k)con.$$

$$
\begin{array}{rcl}
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \uplus \mathrm{RecType} \uplus \mathrm{FunType} \uplus \mathrm{ConsType} \\
(\tau_1, \cdots, \tau_k) \text{ or } \tau^{(k)} & \in & \mathrm{Type}^k \\
(\alpha_1, \cdots, \alpha_k) \text{ or } \alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\varrho & \in & \mathrm{RecType} = \mathrm{Lab} \xrightarrow{\text{fin}} \mathrm{Type} \\
\tau \to \tau' & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \uplus_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \uplus_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \uplus_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
S \text{ or } (m, E) & \in & \mathrm{Str} = \mathrm{StrName} \times \mathrm{Env} \\
(\theta, CE) & \in & \mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ConEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\text{fin}} \mathrm{Str} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\text{fin}} \mathrm{TyStr} \\
CE & \in & \mathrm{ConEnv} = \mathrm{Id} \xrightarrow{\text{fin}} \mathrm{TypeScheme} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Id} \xrightarrow{\text{fin}} \mathrm{TypeScheme} \\
E \text{ or } (SE, TE, VE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
U & \in & \mathrm{TyVarSet} = \mathrm{Fin}(\mathrm{TyVar}) \\
C \text{ or } T, U, E & \in & \mathrm{Context} = \mathrm{TyNameSet} \times \mathrm{TyVarSet} \times \mathrm{Env} \\
\varphi_{\mathrm{Ty}} & \in & \mathrm{TyRea} = \mathrm{TyName} \to \mathrm{TypeFcn} \\
\gamma & \in & \mathrm{Trace} = \mathrm{Tree}(\mathrm{SimTrace} \uplus \mathrm{TraceScheme}) \\
 & & \mathrm{SimTrace} = \mathrm{Type} \uplus \mathrm{Env} \uplus (\mathrm{Context} \times \mathrm{Type}) \uplus \\
 & & \qquad (\mathrm{Context} \times \mathrm{Env}) \uplus \mathrm{TyEnv} \uplus (\mathrm{VarEnv} \times \mathrm{TyRea}) \\
 & & \mathrm{TraceScheme} = \uplus_{k \geq 0} \mathrm{TraceScheme}^{(k)} \\
\forall\alpha^{(k)}.\gamma & \in & \mathrm{TraceScheme}^{(k)} = \mathrm{TyVar}^k \times \mathrm{Trace}
\end{array}
$$

Figure 12: Compound Semantic Objects

Injection: Components may be injected into tuple classes; for example, "VE in Env" means the environment $(\{\}, \{\}, VE)$. The default values for the missing components are always $\emptyset$, $\{\}$, or tuples of these. Injection into disjoint union classes is usually left implicit, with a few exceptions where we also use the "In" notation for it. For brevity, we often express the composition of injections by a single injection.

Modification: The modification of one map f by another map g, written $f + g$, has already been mentioned. It is commonly used for environment modification, for example $E + E'$. Often, empty components will be left implicit in a modification; for example $E + VE$ means $E + (\{\}, \{\}, VE)$. For set components, modification means union, so that $C + (T, VE)$ means

$$(\ (T \text{ of } C) \cup T,\ U \text{ of } C,\ (E \text{ of } C) + VE\ )$$

We frequently need to modify a context C by an environment E (or a type environment TE say), at the same time extending T of C to include the type names of E (or of TE say). We therefore define $C \oplus TE$, for example, to mean $C + (\text{tynames}\, TE, TE)$.

4.4 Types and Type functions

A type τ is an *equality type*, or *admits equality*, if it is of one of the forms

- α, where α admits equality;
- $\{lab_1 \mapsto \tau_1,\ \cdots,\ lab_n \mapsto \tau_n\}$, where each τ_i admits equality;
- $\tau^{(k)} t$, where t and all members of $\tau^{(k)}$ admit equality.

A type function $\theta = \Lambda\alpha^{(k)}.\tau$ has arity k; it must be *closed* – i.e. $\text{tyvars}(\tau) \subseteq \alpha^{(k)}$ – and the bound variables must be distinct. Two type functions are considered equal if they only differ in their choice of bound variables (alpha-conversion). In particular, the equality attribute has no significance in a bound variable of a type function; for example, $\Lambda\alpha.\alpha \to \alpha$ and $\Lambda\beta.\beta \to \beta$ are equal type functions even if α admits equality but β does not. If t has arity k, then we write t to mean $\Lambda\alpha^{(k)}.\alpha^{(k)} t$ (eta-conversion); thus TyName $\subseteq$ TypeFcn. $\theta = \Lambda\alpha^{(k)}.\tau$ is an *equality* type function, or *admits equality*, if when the type variables $\alpha^{(k)}$ are chosen to admit equality then τ also admits equality.

We write the application of a type function θ to a vector $\tau^{(k)}$ of types as $\tau^{(k)}\theta$. If $\theta = \Lambda\alpha^{(k)}.\tau$ we set $\tau^{(k)}\theta = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ (beta-conversion).

We write $\tau\{\theta^{(k)}/t^{(k)}\}$ for the result of substituting type functions $\theta^{(k)}$ for type names $t^{(k)}$ in τ. Here and for other forms of substitution maps, we use the notation $\{t/x\ ;\ \phi\}$ instead of $\{x \mapsto t\ ;\ \phi\}$. We assume that all beta-conversions are carried out after substitution, so that for example

$$(\tau^{(k)} t)\{\Lambda\alpha^{(k)}.\tau/t\} = \tau\{\tau^{(k)}/\alpha^{(k)}\}.$$

4.5 Type Schemes

A *type substitution* is a finite map $\vartheta \in \mathrm{TyVar} \xrightarrow{\text{fin}} \mathrm{Type}$, where $\vartheta(\alpha)$ admits equality if α does, and where $\vartheta(\texttt{num})$ is either `int` or `real` (provided `num` is in the domain of ϑ). Type substitutions extend to functions on other semantic objects, e.g. to functions between types by mapping a type τ to $\tau\{\vartheta(\alpha)/\alpha \ ; \ \alpha \in \mathrm{Dom}\,\vartheta\}$.

A *type scheme* $\sigma = \forall\alpha^{(k)}.\tau$ *generalises* a type τ', written $\sigma \succ \tau'$, if there exists a type substitution ϑ with domain $\alpha^{(k)} \cap \mathrm{tyvars}(\tau)$ such that $\vartheta(\tau) = \tau'$. We make the type substitution of a generalisation explicit by writing $\sigma \succ_\vartheta \tau'$. If $\sigma' = \forall\beta^{(l)}.\tau'$ then σ *generalises* σ', written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and $\beta^{(l)}$ contains no free type variable of σ. It can be shown that $\sigma \succ \sigma'$ iff, for all τ'', whenever $\sigma' \succ \tau''$ then also $\sigma \succ \tau''$.

Two type schemes σ and σ' are considered equal if they can be obtained from each other by renaming and reordering of bound type variables, and deleting type variables from the prefix which do not occur in the body. Here, in contrast to the case for type functions, the equality attribute must be preserved in renaming; for example $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$ are only equal if either both α and β admit equality, or neither does. It can be shown that $\sigma = \sigma'$ iff $\sigma \succ \sigma'$ and $\sigma' \succ \sigma$.

We consider a type τ to be a type scheme, identifying it with $\forall().\tau$.

Traces and Trace Schemes

Traces record objects of the static semantics. The purpose of this recording is twofold: (i) it reifies choices that have been made during static analysis, making it possible to talk about all possible choices for the elaboration of a phrase; and (ii) semantic objects that are involved in the elaboration of a phrase become visible for verification purposes. One can informally view traces as information hung on the syntax tree by static analysis for use by the verification semantics.

Traces can contain bound type variables: in traces of the form $\forall\alpha^{(k)}.\gamma$ the type variables $\alpha^{(k)}$ bind occurrences of those variables in γ. The definition of equality of trace schemes is analogous to that of type schemes.

4.6 Scope of Explicit Type Variables

In the Core language, a type or datatype binding can explicitly introduce type variables whose scope is that binding. In the modules, a description of a value, type, or datatype may contain explicit type variables whose scope is that description. However, we still have to account for the scope of an explicit type variable occurring in the "`:` *ty*" of a typed expression or pattern. For the rest of this section, we consider such occurrences of type variables only.

We call value declarations, axiom declarations and specifications, quantifier expressions and expressions of the form $exp_1^\bullet$ `==` $exp_2^\bullet$ *guarding constructs*. Every occurrence of a guarding construct is said to *scope* a set of explicit type variables determined as follows.

First, an occurrence of α in a guarding construct *phrase* is said to be *unguarded* if the occurrence is not part of a smaller guarding construct within *phrase*. In this case we say that α *occurs unguarded* in the guarding construct.

Then we say that α is *scoped at* a particular occurrence O of a guarding construct in a program if (1) α occurs unguarded in this construct, and (2) α does not occur unguarded in any larger guarding construct containing the occurrence O.

The inference rules in Section 4.10 make this explicit (in contrast to the SML static semantics in [MTH90]). If *phrase* is a guarding construct, then its sentences are typically derived with premises of the form $C + U \vdash \mathit{phrase} \Rightarrow A, U, \gamma$ for some metavariable A. The two occurrences of U mean that every type variable occurring unguarded in *phrase* cannot be locally bound within *phrase*.

4.7 Non-expansive Expressions

Deleted

4.8 Closure

Let τ be a type and A a semantic object. Then $\mathrm{Clos}_A(\tau)$, the *closure* of τ with respect to A, is the type scheme $\forall\alpha^{(k)}.\tau$, where $\alpha^{(k)} = \mathrm{tyvars}(\tau) \setminus \mathrm{tyvars}\, A$. Commonly, A will be a context C. We abbreviate the *total* closure $\mathrm{Clos}_{\{\}}(\tau)$ to $\mathrm{Clos}(\tau)$. If the range of a variable environment VE contains only types (rather than arbitrary type schemes) we set

$$\mathrm{Clos}_A VE = \{id \mapsto \mathrm{Clos}_A(\tau)\ ;\ VE(id) = \tau\}$$

with a similar definition for $\mathrm{Clos}_A CE$.

There is also a similar closure operation for traces: $\mathrm{Clos}_A\gamma = \forall\alpha^{(k)}.\gamma$, where $\alpha^{(k)} = \mathrm{tyvars}(\gamma) \setminus \mathrm{tyvars}\, A$.

4.9 Type Structures and Type Environments

A type structure (θ, CE) is *well-formed* if either $CE = \{\}$, or θ is a type name t. (The latter case arises, with $CE \neq \{\}$, in `datatype` 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 $CE(con)$ of the form $\forall\alpha^{(k)}.(\tau \to \alpha^{(k)}t)$, the type function $\Lambda\alpha^{(k)}.\tau$ also admits equality. (This ensures that the SML equality predicate `=` will be applicable to a constructed value (con, v) of type $\tau^{(k)}t$ only when it is applicable to the value v itself, whose type is $\tau\{\tau^{(k)}/\alpha^{(k)}\}$.) 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.

For any TE of the form

$$TE = \{tycon_i \mapsto (t_i, CE_i) \ ; \ 1 \leq i \leq k\},$$

where no CE_i is the empty map, and for any E we define $\mathrm{Abs_C}(TE, E)$ to be the environment obtained from E and TE as follows. First, let $\mathrm{Abs_C}(TE)$ be the type environment $\{tycon_i \mapsto (t_i, \{\}) \ ; \ 1 \leq i \leq k\}$ in which all constructor environments CE_i have been replaced by the empty map. Let $t'_1, \ldots, t'_k$ be new distinct type names ("new" means that $t'_1, \ldots, t'_k$ are not in $T \mathrm{of} C$), none of which admit equality. Then $\mathrm{Abs_C}(TE, E)$ is the result of simultaneously substituting t'_i for t_i, $1 \leq i \leq k$, throughout $\mathrm{Abs_C}(TE) + E$. (The effect of the latter substitution is to ensure that the use of SML equality on an `abstype` is restricted to the `with` part.) Let φ_{Ty} be the type realisation with $\mathrm{Supp}(\varphi_{\mathrm{Ty}}) \subseteq \{t'_1, \ldots, t'_k\}$ and $\varphi_{\mathrm{Ty}}(t'_i) = t_i$, $1 \leq i \leq k$; we write $\mathrm{Abs_C}(TE, E) =_{\varphi_{\mathrm{Ty}}} E'$ to say that $\mathrm{Abs_C}(TE, E) = E'$ via the type realisation φ_{Ty}. Type realisations are defined in Section 5.6.

4.10 Inference Rules

Each rule of the semantics allows inferences among sentences of the form

$$A \vdash phrase \Rightarrow A'$$

where A is usually a context, *phrase* is a phrase of the Core, and A' is a semantic object — usually a type or an environment or assembly of such objects, together with a trace. It may be pronounced "*phrase* elaborates to A' in (context) A". Some rules have extra hypotheses not of this form; these hypotheses are called *side conditions*.

Atomic Expressions $\boxed{C \vdash atexp \Rightarrow \tau, U, \gamma}$

$$\frac{}{C \vdash scon \Rightarrow \mathrm{type}(scon), \emptyset, \epsilon} \quad (1)$$

$$\frac{C(longvar) \succ \tau}{C \vdash longvar \Rightarrow \tau, \emptyset, \tau} \quad (2)$$

$$\frac{C(longcon) \succ \tau}{C \vdash longcon \Rightarrow \tau, \emptyset, \tau} \quad (3)$$

$$\frac{C(longexcon) = \tau}{C \vdash longexcon \Rightarrow \tau, \emptyset, \epsilon} \quad (4)$$

$$\frac{\langle C \vdash exprow \Rightarrow \varrho, U, \gamma \rangle}{C \vdash \{ \ \langle exprow \rangle \ \} \Rightarrow \{\}\langle + \ \varrho \rangle \text{ in Type}, \ \emptyset\langle \cup \mathrm{U} \rangle, \ \epsilon\langle \cdot \gamma \rangle} \quad (5)$$

$$\frac{C \vdash dec \Rightarrow E, \gamma \qquad C \oplus E \vdash exp \Rightarrow \tau, U, \gamma' \qquad \text{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau, U, \gamma \cdot \gamma'} \tag{6}$$

$$\frac{C \vdash exp \Rightarrow \tau, U, \gamma}{C \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow \tau, U, \gamma} \tag{7}$$

$$\frac{}{C \vdash \texttt{?} \Rightarrow \tau, \emptyset, (C, \tau)} \tag{7.1}$$

Comments:

(2),(3) The instantiation of type schemes allows different occurrences of a single *longvar* or *longcon* to assume different types.

(6) The use of $\oplus$, here and elsewhere, ensures that type names generated by the first sub-phrase are different from type names generated by the second sub-phrase.

The third premise is not present in the SML definition. Simply omitting it would compromise the soundness of type inference, because type names introduced by different `let` expressions could become equal. This was an oversight in the definition of SML [MTH90] which was not fixed in [MT91]; see also [Kah93]. Some Standard ML implementations have a less restrictive method for type-checking `let`-expressions which is still sound but allows local datatypes to escape from the scope of the `let`.

Notice that there are no unguarded occurrences of explicit type variables in declarations, as the form of sentences for declarations indicates. This differs from SML, which permits (unguarded) imperative type variables in exception declarations.

(7.1) A `?` can have *any* type. The context C is stored in the trace to enable verification to type-check the chosen replacement for `?` in a given model, see rule 201.

Expression Rows

$\boxed{C \vdash exprow \Rightarrow \varrho, U, \gamma}$

$$\frac{C \vdash exp \Rightarrow \tau, U, \gamma \qquad \langle C \vdash exprow \Rightarrow \varrho, U', \gamma' \rangle}{C \vdash lab\ \texttt{=}\ exp\ \langle\ \texttt{,}\ exprow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle,\ U \langle \cup U' \rangle,\ \gamma \langle \cdot \gamma' \rangle} \tag{8}$$

Expressions

$\boxed{C \vdash exp \Rightarrow \tau, U, \gamma}$

$$\frac{C \vdash atexp \Rightarrow \tau, U, \gamma}{C \vdash atexp \Rightarrow \tau, U, \gamma} \tag{9}$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau, U, \gamma \qquad C \vdash atexp \Rightarrow \tau', U', \gamma'}{C \vdash exp\ atexp \Rightarrow \tau, U \cup U', \gamma \cdot \gamma'} \tag{10}$$

$$\frac{C \vdash exp \Rightarrow \tau, U, \gamma \qquad C \vdash ty \Rightarrow \tau \qquad U' = \text{tyvars}\, ty}{C \vdash exp\ :\ ty \Rightarrow \tau, U \cup U', \gamma} \tag{11}$$

$$\frac{C + U \vdash exp_1^\bullet \Rightarrow \tau, U_1, \gamma \qquad C + U \vdash exp_2^\bullet \Rightarrow \tau, U_2, \gamma' \qquad U = U_1 \cup U_2}{C \vdash exp_1^\bullet \texttt{ == } exp_2^\bullet \Rightarrow \texttt{bool}, \emptyset, \text{Clos}_C((C, \tau) \cdot \gamma \cdot \gamma')} \tag{11.1}$$

$$\frac{C + U \vdash match^\bullet \Rightarrow \tau \to \texttt{bool}, U, \gamma}{C \vdash \texttt{exists}\ match^\bullet \Rightarrow \texttt{bool}, \emptyset, \text{Clos}_C((C, \tau) \cdot \gamma)} \tag{11.2}$$

$$\frac{C + U \vdash match^\bullet \Rightarrow \tau \to \texttt{bool}, U, \gamma}{C \vdash \texttt{forall}\ match^\bullet \Rightarrow \texttt{bool}, \emptyset, \text{Clos}_C((C, \tau) \cdot \gamma)} \tag{11.3}$$

$$\frac{C \vdash exp^\bullet \Rightarrow \tau, U, \gamma}{C \vdash exp^\bullet\ \texttt{terminates} \Rightarrow \texttt{bool}, U, \epsilon} \tag{11.4}$$

$$\frac{C \vdash exp \Rightarrow \tau, U, \gamma \qquad C \vdash match \Rightarrow \texttt{exn} \to \tau, U', \gamma'}{C \vdash exp\ \texttt{handle}\ match \Rightarrow \tau, U \cup U', \gamma \cdot \gamma'} \tag{12}$$

$$\frac{C \vdash exp \Rightarrow \texttt{exn}, U, \gamma}{C \vdash \texttt{raise}\ exp \Rightarrow \tau, U, \tau \cdot \gamma} \tag{13}$$

$$\frac{C \vdash match \Rightarrow \tau, U, \gamma}{C \vdash \texttt{fn}\ match \Rightarrow \tau, U, (C, \tau) \cdot \gamma} \tag{14}$$

Comments:

(9) The relation symbol $\vdash$ is overloaded for all syntactic classes (here atomic expressions and expressions). Thus, the relational symbol $\vdash$ refers in the premise to the predicate for atomic expressions as defined in rules 1 to 7.1.

(11) Here τ is determined by C and ty. Notice that type variables in ty cannot be instantiated in obtaining τ; thus the expression `1:'a` will not elaborate successfully, nor will the expression `(fn x=>x):'a->'b`. The effect of type variables in an explicitly typed expression is to indicate exactly the degree of polymorphism present in the expression.

(11.1)–(11.3) Equality and quantification are *guarding constructs*, i.e. any type variable occurring in them is guarded. Any type variable occuring unguarded within a guarding construct is scoped at that construct, provided it is not already scoped in the context: this is expressed by extending the context with $+U$ which prohibits these type variables from being bound locally by the closure operator.

Storing C and τ in the trace is necessary for the verification semantics, to select appropriate witnesses for the quantifiers, and to compare values. We require quantification and comparison in the verification semantics to behave uniformly: abstracting type variables via Clos_C here allows the verification semantics to instantiate the traces arbitrarily.

(11.4) The expression $exp^\bullet$ will not be subject to verification; only its termination behaviour in the dynamic semantics is of interest. The dynamic semantics does not need any type information, thus the empty trace ϵ in the result.

(13) Note that τ does not occur in the premise; thus a `raise` expression has "arbitrary" type. For the same reason, its type has to be recorded in the trace.

(14) The context C is part of the result trace so that *match* can be type-checked after replacing all its question marks with the corresponding choices of a given model.

Matches

$$\boxed{C \vdash match \Rightarrow \tau, U, \gamma}$$

$$\frac{C \vdash mrule \Rightarrow \tau, U, \gamma \qquad \langle C \vdash match \Rightarrow \tau, U', \gamma' \rangle}{C \vdash mrule \ \langle \ \mathtt{|} \ match \rangle \Rightarrow \tau, U\langle \cup U' \rangle, \gamma\langle \cdot \gamma' \rangle} \tag{15}$$

Match Rules

$$\boxed{C \vdash mrule \Rightarrow \tau, U, \gamma}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau), U, \gamma \qquad C \oplus VE \vdash exp \Rightarrow \tau', U', \gamma'}{C \vdash pat \ \mathtt{=>} \ exp \ \Rightarrow \tau \to \tau', \ U \cup U', \ \gamma \cdot \gamma'} \tag{16}$$

Comment: This rule allows new free type variables to enter the context. These new type variables will be chosen, in effect, during the elaboration of *pat* (i.e., in the inference of the first hypothesis). In particular, their choice may have to be made to agree with type variables present in any explicit type expression occurring within *exp* (see rule 11).

Notice that rule 16 uses $C \oplus VE$ in contrast to the SML definition, which uses $C + VE$ here. The reason for this change is another soundness problem, similar to the one mentioned for rule 6, but a bit more involved. This problem was not mentioned in [MT91] or [Kah93], see [Kah94].

Declarations

$\boxed{C \vdash dec \Rightarrow E, \gamma}$

$$\frac{C + U \vdash valbind \Rightarrow VE, U, \gamma \qquad VE' = \mathrm{Clos}_C VE}{C \vdash \texttt{val}\ valbind \Rightarrow VE' \text{ in Env}, \mathrm{Clos}_C \gamma} \tag{17}$$

$$\frac{C \vdash typbind \Rightarrow TE, \gamma}{C \vdash \texttt{type}\ typbind \Rightarrow TE \text{ in Env}, TE \cdot \gamma} \tag{18}$$

$$\frac{C \vdash typbind \Rightarrow TE, \gamma \qquad \forall (\theta, CE) \in \mathrm{Ran}\, TE,\ \theta \text{ admits equality}}{C \vdash \texttt{eqtype}\ typbind \Rightarrow TE \text{ in Env}, TE \cdot \gamma} \tag{18.1}$$

$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE, \gamma \qquad \forall (t, CE) \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ E = (VE, TE) \text{ in Env} \qquad TE \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ datbind \Rightarrow E,\ E \cdot \gamma} \tag{19}$$

$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE, \gamma \qquad \forall (t, CE) \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ C \oplus (VE, TE) \vdash dec \Rightarrow E, \gamma' \qquad TE \text{ maximises equality} \\ \mathrm{Abs}_{\mathrm{C}}(TE, E) =_{\varphi_{\mathrm{Ty}}} E' \end{array}}{C \vdash \texttt{abstype}\ datbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow E',\ (VE, \varphi_{\mathrm{Ty}}) \cdot \gamma \cdot \gamma'} \tag{20}$$

$$\frac{C \vdash exbind \Rightarrow VE, \gamma}{C \vdash \texttt{exception}\ exbind \Rightarrow VE \text{ in Env},\ \gamma} \tag{21}$$

$$\frac{C \vdash dec_1 \Rightarrow E_1, \gamma_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2, \gamma_2}{C \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2} \tag{22}$$

$$\frac{C(longstrid_1) = (m_1, E_1) \quad \cdots \quad C(longstrid_n) = (m_n, E_n)}{C \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n, \epsilon} \tag{23}$$

$$\frac{}{C \vdash \qquad \Rightarrow \{\} \text{ in Env}, \epsilon} \tag{24}$$

$$\frac{C \vdash dec_1 \Rightarrow E_1, \gamma_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2, \gamma_2}{C \vdash dec_1\ \langle ; \rangle\ dec_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2} \tag{25}$$

Comments:

(17) Here VE will contain types rather than general type schemes. The closure of VE is exactly what allows variables to be used polymorphically, via rule 2.

(18) Within *typbind* there might be ?-types, i.e. there might be newly introduced type names. Notice that their equality attributes are not affected by this rule, but that principality of environments can affect it later (rule 57) — a type with equality attribute is more specific.

(18),(18.1) The verification semantics exploits type information; storing the type environment TE in the trace is one way to make the needed information accessible for later verification purposes.

(19),(20) The side conditions express that the elaboration of each datatype binding generates new type names and that as many of these new names as possible admit equality. Adding TE to the context on the left of the $\vdash$ in the first premise captures the recursive nature of the binding.

To see why an environment E etc. is stored in the trace, one has to look at these rules in connection with the corresponding rules of the verification semantics, here 241 and 242.

(20) The Abs_C operation was defined in Section 4.9, page 28.

(21) No closure operation is used here, since exception bindings do not contain type variables.

Value Bindings

$\boxed{C \vdash \mathit{valbind} \Rightarrow VE, U, \gamma}$

$$\frac{\begin{array}{c} C \vdash \mathit{pat} \Rightarrow (VE, \tau), U, \gamma \qquad C \vdash \mathit{exp} \Rightarrow \tau, U', \gamma' \\ \langle C \vdash \mathit{valbind} \Rightarrow VE', U'', \gamma'' \rangle \end{array}}{C \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \Rightarrow VE\ \langle + VE' \rangle,\ U \cup U' \langle \cup U'' \rangle,\ \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle} \tag{26}$$

$$\frac{C + VE \vdash \mathit{valbind} \Rightarrow VE, U, \gamma}{C \vdash \texttt{rec}\ \mathit{valbind} \Rightarrow VE, U, \gamma} \tag{27}$$

Comments:

(26) When the option is present we have $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$ by the syntactic restrictions.

(27) Modifying C by VE on the left of the premise captures the recursive nature of the binding. From rule 26 we see that any type scheme occurring in VE will have to be a type. Thus each use of a recursive function in its own body must be ascribed the same type.

Type Bindings

$\boxed{C \vdash \mathit{typbind} \Rightarrow TE, \gamma}$

$$\frac{\mathit{tyvarseq} = \alpha^{(k)} \qquad C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{typbind} \Rightarrow TE, \gamma \rangle}{\begin{array}{l} C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{typbind} \rangle \Rightarrow \\ \qquad \{\mathit{tycon} \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}\ \langle + TE \rangle,\ \epsilon \langle \cdot \gamma \rangle \end{array}} \tag{28}$$

$$\frac{\begin{array}{c} \mathit{tyvarseq} = \alpha^{(k)} \qquad t \notin (T \text{ of } C),\ \mathrm{arity}\, t = k \\ \langle C + \{t\} \vdash \mathit{typbind} \Rightarrow TE, \gamma \rangle \end{array}}{\begin{array}{l} C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ \mathit{typbind} \rangle \Rightarrow \\ \qquad \{\mathit{tycon} \mapsto (t, \{\})\}\ \langle + TE \rangle,\ (C, t) \langle \cdot \gamma \rangle \end{array}} \tag{28.1}$$

Comments:

(28) The syntactic restrictions ensure that the type function $\Lambda\alpha^{(k)}.\tau$ satisfies the well-formedness constraints of Section 4.4 and that $tycon \notin \mathrm{Dom}\, TE$.

(28.1) Question mark types are treated as new types for the purposes of static analysis (premise $t \notin (T \text{ of } C)$). In a given model (verification semantics), the type name t will be replaced by an appropriate type function, see rule 252; C and T are stored in the trace for the purposes of that replacement.

Data Type Bindings

$\boxed{C \vdash datbind \Rightarrow VE, TE, \gamma}$

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)} t \vdash conbind \Rightarrow CE, \gamma \\ \langle C \vdash datbind \Rightarrow VE, TE, \gamma' \qquad \forall (t', CE) \in \mathrm{Ran}\, TE, t \neq t' \rangle \end{array}}{\begin{array}{c} C \vdash tyvarseq\ tycon\ \texttt{=}\ conbind\ \langle \texttt{and}\ datbind \rangle \Rightarrow \\ \mathrm{Clos}\, CE\ \langle + \ VE \rangle,\ \{tycon \mapsto (t, \mathrm{Clos}\, CE)\}\ \langle + \ TE \rangle,\ \gamma \langle \cdot \gamma' \rangle \end{array}} \tag{29}$$

Comment: The syntactic restrictions ensure $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, CE = \emptyset$ and $tycon \notin \mathrm{Dom}\, TE$.

Constructor Bindings

$\boxed{C, \tau \vdash conbind \Rightarrow CE, \gamma}$

$$\frac{con = id^{\mathsf{C}} \qquad \langle C, \tau \vdash conbind \Rightarrow CE, \gamma \rangle}{C, \tau \vdash con\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow \{id \mapsto \tau\}\ \langle + \ CE \rangle,\ \tau \langle \cdot \gamma \rangle} \tag{30}$$

$$\frac{\begin{array}{c} con = id^{\mathsf{C}} \qquad C \vdash ty \Rightarrow \tau' \qquad \tau'' = \tau' \to \tau \\ \langle C, \tau \vdash conbind \Rightarrow CE, \gamma \rangle \end{array}}{C, \tau \vdash con\ \texttt{of}\ ty\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow \{id \mapsto \tau''\}\ \langle + \ CE \rangle,\ \tau'' \langle \cdot \gamma \rangle} \tag{30.1}$$

Comments:

(30),(30.1) By the syntactic restrictions $con \notin \mathrm{Dom}\, CE$.

Exception Bindings

$\boxed{C \vdash exbind \Rightarrow VE, \gamma}$

$$\frac{excon = id^{\mathsf{e}} \qquad \langle C \vdash exbind \Rightarrow VE, \gamma \rangle}{C \vdash excon\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{id \mapsto \texttt{exn}\}\ \langle + \ VE \rangle,\ \epsilon \langle \cdot \gamma \rangle} \tag{31}$$

$$\frac{\begin{array}{c} excon = id^{\mathsf{e}} \qquad C \vdash ty \Rightarrow \tau \qquad \mathrm{tyvars}(ty) = \emptyset \\ \langle C \vdash exbind \Rightarrow VE, \gamma \rangle \end{array}}{C \vdash excon\ \texttt{of}\ ty\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{id \mapsto \tau \to \texttt{exn}\}\ \langle + \ VE \rangle,\ \tau \langle \cdot \gamma \rangle} \tag{31.1}$$

$$\frac{excon = id^{\mathtt{e}} \qquad C(longexcon) = \tau \qquad \langle C \vdash exbind \Rightarrow VE, \gamma \rangle}{C \vdash excon \texttt{ = } longexcon\ \langle \texttt{and } exbind \rangle \Rightarrow \{id \mapsto \tau\}\ \langle + VE \rangle,\ \epsilon \langle \cdot \gamma \rangle} \tag{32}$$

Comments:

(31.1) Notice that *ty* must not contain any type variables. This is slightly stricter than to require tyvars $\tau = \emptyset$, as the corresponding rule in [MTH90] does, and it rules out some pathological cases. Type variables occurring in exception bindings are unguarded (in SML) and affect the scoping mechanism even if they do not occur in the type obtained from that binding. In Extended ML, declarations never have unguarded type variables; the restriction makes sure that there is no difference in type variable scoping between Standard ML and Extended ML.

(31),(31.1),(32) There are unique VE and γ, for each C and *exbind*, such that $C \vdash exbind \Rightarrow VE, \gamma$.

Atomic Patterns

$$\boxed{C \vdash atpat \Rightarrow (VE, \tau), U, \gamma}$$

$$\frac{}{C \vdash _ \Rightarrow (\{\}, \tau), \emptyset, \tau} \tag{33}$$

$$\frac{}{C \vdash scon \Rightarrow (\{\}, \mathrm{type}(scon)), \emptyset, \epsilon} \tag{34}$$

$$\frac{var = id^{\mathtt{v}}}{C \vdash var \Rightarrow (\{id \mapsto \tau\}, \tau), \emptyset, \tau} \tag{35}$$

$$\frac{C(longcon) \succ \tau^{(k)}t}{C \vdash longcon \Rightarrow (\{\}, \tau^{(k)}t), \emptyset, \tau^{(k)}t} \tag{36}$$

$$\frac{C(longexcon) = \texttt{exn}}{C \vdash longexcon \Rightarrow (\{\}, \texttt{exn}), \emptyset, \epsilon} \tag{37}$$

$$\frac{\langle C \vdash patrow \Rightarrow (VE, \varrho), U, \gamma \rangle}{C \vdash \texttt{\{}\ \langle patrow \rangle\ \texttt{\}} \Rightarrow (\ \{\}\langle + VE \rangle,\ \{\}\langle + \varrho \rangle \text{ in Type}),\ \emptyset \langle \cup U \rangle,\ \epsilon \langle \cdot \gamma \rangle} \tag{38}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau), U, \gamma}{C \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow (VE, \tau), U, \gamma} \tag{39}$$

Comments:

(35) Note that *var* can assume a type, not a general type scheme.

Pattern Rows

$\boxed{C \vdash patrow \Rightarrow (VE, \varrho), U, \gamma}$

$$\frac{}{C \vdash \ldots \Rightarrow (\{\}, \varrho), \emptyset, \varrho \text{ in Trace}} \tag{40}$$

$$\frac{\begin{array}{c} C \vdash pat \Rightarrow (VE, \tau), U, \gamma \\ \langle C \vdash patrow \Rightarrow (VE', \varrho), U', \gamma' \qquad lab \notin \mathrm{Dom}\, \varrho \rangle \end{array}}{\begin{array}{c} C \vdash lab \texttt{ = } pat \langle \texttt{ , } patrow \rangle \Rightarrow \\ (VE \langle + VE' \rangle,\ \{lab \mapsto \tau\} \langle + \varrho \rangle),\ U \langle \cup U' \rangle,\ \gamma \langle \cdot \gamma' \rangle \end{array}} \tag{41}$$

Comments:

(41) By the syntactic restrictions, $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$.

Patterns

$\boxed{C \vdash pat \Rightarrow (VE, \tau), U, \gamma}$

$$\frac{C \vdash atpat \Rightarrow (VE, \tau), U, \gamma}{C \vdash atpat \Rightarrow (VE, \tau), U, \gamma} \tag{42}$$

$$\frac{C(longcon) \succ \tau'' = \tau' \to \tau \qquad C \vdash atpat \Rightarrow (VE, \tau'), U, \gamma}{C \vdash longcon\ atpat \Rightarrow (VE, \tau), U, \tau'' \cdot \gamma} \tag{43}$$

$$\frac{C(longexcon) = \tau \to \texttt{exn} \qquad C \vdash atpat \Rightarrow (VE, \tau), U, \gamma}{C \vdash longexcon\ atpat \Rightarrow (VE, \texttt{exn}), U, \gamma} \tag{44}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau), U, \gamma \qquad C \vdash ty \Rightarrow \tau \qquad U' = \mathrm{tyvars}\, ty}{C \vdash pat\ :\ ty \Rightarrow (VE, \tau), U \cup U', \gamma} \tag{45}$$

$$\frac{\begin{array}{c} C \vdash var \Rightarrow (VE, \tau), U, \gamma \qquad C \vdash pat \Rightarrow (VE', \tau), U', \gamma' \\ \langle C \vdash ty \Rightarrow \tau \qquad U'' = \mathrm{tyvars}\, ty \rangle \end{array}}{C \vdash var \langle :\ ty \rangle\ \texttt{as}\ pat \Rightarrow (VE + VE', \tau), U \cup U' \langle \cup U'' \rangle, \gamma \cdot \gamma'} \tag{46}$$

Comments:

(46) By the syntactic restrictions, $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$. In the first premise, *var* is viewed as an atomic pattern.

Type Expressions

$\boxed{C \vdash ty \Rightarrow \tau}$

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \tag{47}$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \texttt{\{}\ \langle tyrow \rangle\ \texttt{\}} \Rightarrow \{\} \langle + \varrho \rangle \text{ in Type}} \tag{48}$$

$$\frac{tyseq = ty_1 \cdots ty_k \qquad C \vdash ty_i \Rightarrow \tau_i \ (1 \le i \le k) \qquad C(longtycon) = (\theta, CE) \qquad \text{arity } \theta = k}{C \vdash tyseq\ longtycon \Rightarrow \tau^{(k)}\theta} \tag{49}$$

$$\frac{C \vdash ty \Rightarrow \tau \qquad C \vdash ty' \Rightarrow \tau'}{C \vdash ty\ \texttt{->}\ ty' \Rightarrow \tau \to \tau'} \tag{50}$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash \texttt{(}\ ty\ \texttt{)} \Rightarrow \tau} \tag{51}$$

Type-expression Rows

$\boxed{C \vdash tyrow \Rightarrow \varrho}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab\ \texttt{:}\ ty\ \langle\ \texttt{,}\ tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle} \tag{52}$$

Comment: The syntactic constraints ensure $lab \notin \mathrm{Dom}\ \varrho$.

4.11 Further Restrictions

In a match of the form pat_1 `=>` exp_1 `|` $\cdots$ `|` $\texttt{pat}_n$ `=>` exp_n the pattern sequence $pat_1, \ldots, pat_n$ should be *irredundant*; that is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$. In the context `fn` *match*, the *match* must also be *exhaustive*; that is, every value (of the right type) must be matched by some pat_i. The compiler must give warning on violation of these restrictions, but should still compile the match. The restrictions are inherited by derived forms; in particular, this means that in the function binding *var* $atpat_1 \cdots atpat_n \langle$`:` *ty*$\rangle$ `=` *exp* (consisting of one clause only), each separate $atpat_i$ should be exhaustive by itself.

This text originates from [MTH90]. In the context of Extended ML, this (and other references to compilers below) should be taken as referring to an Extended ML parser/typechecker, which of course is not a compiler in the usual sense.

5 Static Semantics for Modules

5.1 Semantic Objects

The simple objects for Modules static semantics are exactly as for the Core. The compound objects are those for the Core, augmented by those in Figure 13.

$$
\begin{array}{rcl}
M & \in & \text{StrNameSet} = \text{Fin}(\text{StrName}) \\
N \text{ or } (M,T) & \in & \text{NameSet} = \text{StrNameSet} \times \text{TyNameSet} \\
\Sigma \text{ or } (N)S & \in & \text{Sig} = \text{NameSet} \times \text{Str} \\
\Phi \text{ or } (N)(S,(N')S') & \in & \text{FunSig} = \text{NameSet} \times (\text{Str} \times \text{Sig}) \\
G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\
F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunSig} \\
B \text{ or } N,F,G,E & \in & \text{Basis} = \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
\varphi_{\text{Str}} & \in & \text{StrRea} = \text{StrName} \to \text{StrName} \\
\varphi \text{ or } (\varphi_{\text{Ty}}, \varphi_{\text{Str}}) & \in & \text{Rea} = \text{TyRea} \times \text{StrRea} \\
\gamma & \in & \text{Trace} = \text{Tree}(\text{SimTrace} \uplus \text{TraceScheme} \uplus \text{BoundTrace}) \\
(N)\gamma & \in & \text{BoundTrace} = \text{NameSet} \times \text{Trace} \\
 & & \text{SimTrace} = \text{SimTrace}_{\text{COR}} \uplus \text{StrName} \uplus \text{Rea} \uplus \text{VarEnv}
\end{array}
$$

Figure 13: Further Compound Semantic Objects

The prefix (N), in signatures and functor signatures, binds both type names and structure names. We shall always consider a set N of names as partitioned into a pair (M,T) of sets of the two kinds of name.

It is sometimes convenient to work with an arbitrary semantic object A, or *assembly* A of such objects. As with the function tynames, strnames(A) and names(A) denote respectively the set of structure names and the set of names occurring free in A.

Certain operations require a change of bound names in semantic objects; see for example Section 5.7. When bound type names are changed, we demand that all of their attributes (i.e. equality and arity) are preserved.

For any structure $S = (m,(SE,TE,VE))$ we call m the *structure name* or *name* of S; also, the *proper substructures* of S are the members of $\text{Ran}\, SE$ and their proper substructures. The *substructures* of S are S itself and its proper substructures. The structures *occurring in* an object or assembly A are the structures and substructures from which it is built. The type structures of S are all members of $\text{Ran}\, TE$ and all type structures of substructures of S.

The operations of projection, injection and modification are as for the Core. Moreover, we define $C \text{ of } B$ to be the context $(T \text{ of } B, \emptyset, E \text{ of } B)$, i.e. with an empty set of explicit type variables. Also, we frequently need to modify a basis B by an environment E (or a structure environment SE say), at the same time extending

N of B to include the type names and structure names of E (or of SE say). We therefore define $B \oplus SE$, for example, to mean $B +$ (names SE, SE).

For the purposes of the module semantics, we extend the notion of simple trace and introduce another form of trace, the *bound trace*. $\text{SimTrace}_{\text{COR}}$ refers to SimTrace as defined in Figure 12 on page 24, i.e. the simple traces of the Core.

In traces of the form $(N)\gamma$, the names N are bound. Type substitution and realisations have to respect variable binding when applied to a trace, i.e. their application may involve renaming of bound names and bound type variables. When applying a realisation φ to a trace consisting of a realisation φ', then φ affects the results of φ', i.e. $\varphi(\varphi')(x) = \varphi(\varphi'(x))$.

5.2 Consistency

A set of type structures is said to be *consistent* if, for all (θ_1, CE_1) and (θ_2, CE_2) in the set, if $\theta_1 = \theta_2$ then

$$CE_1 = \{\} \text{ or } CE_2 = \{\} \text{ or } \text{Dom}\, CE_1 = \text{Dom}\, CE_2$$

A semantic object A or assembly A of objects is said to be *consistent* if (after changing bound names to make all nameset prefixes in A disjoint) for all S_1 and S_2 occurring in A and for every *longstrid* and every *longtycon*

1. If m of $S_1 = m$ of S_2, and both $S_1(\mathit{longstrid})$ and $S_2(\mathit{longstrid})$ exist, then

$$m \text{ of } S_1(\mathit{longstrid}) \;=\; m \text{ of } S_2(\mathit{longstrid})$$

2. If m of $S_1 = m$ of S_2, and both $S_1(\mathit{longtycon})$ and $S_2(\mathit{longtycon})$ exist, then

$$\theta \text{ of } S_1(\mathit{longtycon}) \;=\; \theta \text{ of } S_2(\mathit{longtycon})$$

3. The set of all type structures in A is consistent

As an example, a functor signature $(N)(S, (N')S')$ is consistent if, assuming first that $N \cap N' = \emptyset$, the assembly $A = \{S, S'\}$ is consistent.

We may loosely say that two structures S_1 and S_2 are consistent if $\{S_1, S_2\}$ is consistent, but must remember that this is stronger than the assertion that S_1 is consistent and S_2 is consistent.

Note that if A is a consistent assembly and $A' \subset A$ then A' is also a consistent assembly.

5.3 Well-formedness

A signature $(N)S$ is *well-formed* if $N \subseteq$ names S, and also, whenever (m, E) is a substructure of S and $m \notin N$, then $N \cap$ (names E) $= \emptyset$, and whenever (t, CE) is a type structure of S and $t \notin N$, then $N \cap$ (names CE) $= \emptyset$. A functor

signature $(N)(S, (N')S')$ is *well-formed* if $(N)S$ and $(N')S'$ are well-formed, and also, whenever (m', E') is a substructure of S' and $m' \notin N \cup N'$, then $(N \cup N') \cap (\text{names}\, E') = \emptyset$, and whenever (t', CE') is a type structure of S' and $t' \notin N \cup N'$, then $(N \cup N') \cap (\text{names}\, CE') = \emptyset$.

An object or assembly A is *well-formed* if every signature and functor signature occurring in A is well-formed.

5.4 Cycle-freedom

An object or assembly A is *cycle-free* if it contains no cycle of structure names; that is, there is no sequence

$$m_0, \cdots, m_{k-1}, m_k = m_0 \quad (k > 0)$$

of structure names such that, for each i $(0 \leq i < k)$ some structure with name m_i occurring in A has a proper substructure with name m_{i+1}.

5.5 Admissibility

An object or assembly A is *admissible* if it is consistent, well-formed and cycle-free. Henceforth it is assumed that all objects mentioned are admissible. We also require that

1. In every sentence $A \vdash \mathit{phrase} \Rightarrow A'$ inferred by the rules given in Section 5.14, the assembly $\{A, A'\}$ is admissible.

2. In the special case of a sentence $B \vdash \mathit{sigexp} \Rightarrow S, \gamma$, we further require that the assembly consisting of all semantic objects occurring in the entire inference of this sentence be admissible. This is important for the definition of principal signatures in Section 5.13.

In our semantic definition we have not undertaken to indicate how admissibility should be checked in an implementation.

5.6 Type Realisation

A *type realisation* is a function $\varphi_{\text{Ty}} : \text{TyName} \to \text{TypeFcn}$ such that t and $\varphi_{\text{Ty}}(t)$ have the same arity, and if t admits equality then so does $\varphi_{\text{Ty}}(t)$.

The *support* $\text{Supp}\, \varphi_{\text{Ty}}$ of a type realisation φ_{Ty} is the set of type names t for which $\varphi_{\text{Ty}}(t) \neq t$.

5.7 Realisation

A *realisation* is a function φ of names, partitioned into a type realisation $\varphi_{\text{Ty}} : \text{TyName} \to \text{TypeFcn}$ and a function $\varphi_{\text{Str}} : \text{StrName} \to \text{StrName}$. The *support*

$\mathrm{Supp}\,\varphi$ of a realisation φ is the set of names n for which $\varphi(n) \neq n$. The *yield* $\mathrm{Yield}\,\varphi$ of a realisation φ is the set of names which occur in some $\varphi(n)$ for which $n \in \mathrm{Supp}\,\varphi$.

Realisations φ are extended to apply to all semantic objects; their effect is to replace each name n by $\varphi(n)$. In applying φ to an object with bound names, such as a signature $(N)S$, first bound names must be changed so that, for each binding prefix (N),

$$N \cap (\mathrm{Supp}\,\varphi \cup \mathrm{Yield}\,\varphi) = \emptyset \ .$$

We assume realisations to have finite support, i.e. the last sentence does not affect the applicability of realisations.

The semantic class of realisations is called Rea.

Traces

We extend the definition of generalisation to traces. The relation $\succ$ is the smallest binary relation on traces satisfying the following properties:

$$\begin{array}{lcll}
\gamma & \succ & \gamma & \Longleftarrow \quad \gamma \in \mathrm{SimTrace} \\
\gamma_1 \cdot \gamma_2 & \succ & \gamma_1' \cdot \gamma_2' & \Longleftarrow \quad \gamma_1 \succ \gamma_1' \ \wedge \ \gamma_2 \succ \gamma_2' \\
\forall \alpha^{(k)}.\gamma & \succ & \vartheta(\gamma) & \Longleftarrow \quad \mathrm{Dom}\,\vartheta = \alpha^{(k)} \\
(N)\gamma & \succ & \varphi(\gamma) & \Longleftarrow \quad \mathrm{Supp}\,\varphi \subseteq N \\
\gamma_1 & \succ & \gamma_3 & \Longleftarrow \quad \text{there exists } \gamma_2 \text{ such that } \gamma_1 \succ \gamma_2 \ \wedge \gamma_2 \succ \gamma_3 \\
\gamma_1 & \succ & \gamma_2 & \Longleftarrow \quad \text{for all } \gamma_3, \gamma_2 \succ \gamma_3 \implies \gamma_1 \succ \gamma_3
\end{array}$$

It can be shown that generalisation between type schemes is a special case of generalisation between traces.

Stripping Axioms

Axioms should not influence elaboration; in particular, the successful elaboration of a phrase should not depend on the presence of axioms. For this purpose, we define a family of partial functions

$$\mathrm{strip} : \begin{cases} (\mathrm{SigExp} \times \mathrm{Trace}) \to (\mathrm{SigExp} \times \mathrm{Trace}) \\ (\mathrm{StrDesc} \times \mathrm{Trace}) \to (\mathrm{StrDesc} \times \mathrm{Trace}) \\ (\mathrm{Spec} \times \mathrm{Trace}) \to (\mathrm{Spec} \times \mathrm{Trace}) \end{cases}$$

which "strip" axioms from signature expressions and perform the corresponding removal in the trace. These functions are partial because e.g. the trace γ in an argument $(spec, \gamma)$ of strip is expected to be an elaboration result of *spec*. More precisely, strip is defined as follows:

$$\begin{array}{l}
\mathrm{strip} : (\mathrm{SigExp} \times \mathrm{Trace}) \to (\mathrm{SigExp} \times \mathrm{Trace}) \\
\mathrm{strip}(\texttt{sig}\ spec\ \texttt{end}, m \cdot \gamma) \ = \ (\texttt{sig}\ spec'\ \texttt{end}, m \cdot \gamma') \\
\qquad \mathit{where}\ \mathrm{strip}(spec, \gamma) = (spec', \gamma') \\
\mathrm{strip}(sigid, \gamma) \ = \ (sigid, \gamma)
\end{array}$$

$\mathrm{strip} : (\mathrm{StrDesc} \times \mathrm{Trace}) \to (\mathrm{StrDesc} \times \mathrm{Trace})$
$\mathrm{strip}(\mathit{strid} : \mathit{sigexp}\ \langle \mathtt{and}\ \mathit{strdesc}\rangle, \gamma_1\langle\cdot\gamma_2\rangle) =$
$\quad (\mathit{strid} : \mathit{sigexp}'\ \langle \mathtt{and}\ \mathit{strdesc}'\rangle, \gamma_1'\langle\cdot\gamma_2'\rangle)$
$\quad \mathit{where} \begin{cases} \mathrm{strip}(\mathit{sigexp}, \gamma_1) = (\mathit{sigexp}', \gamma_1') \\ \langle \mathrm{strip}(\mathit{strdesc}, \gamma_2) = (\mathit{strdesc}', \gamma_2')\rangle \end{cases}$

$\mathrm{strip} : (\mathrm{Spec} \times \mathrm{Trace}) \to (\mathrm{Spec} \times \mathrm{Trace})$
$\mathrm{strip}(\mathtt{axiom}\ \mathit{axdesc}, \gamma) = (\quad, \epsilon)$
$\mathrm{strip}(\mathtt{structure}\ \mathit{strdesc}, \gamma) = (\mathtt{structure}\ \mathit{strdesc}', \gamma')$
$\quad \mathit{where}\ \mathrm{strip}(\mathit{strdesc}, \gamma) = (\mathit{strdesc}', \gamma')$
$\mathrm{strip}(\mathtt{local}\ \mathit{spec}_1\ \mathtt{in}\ \mathit{spec}_2\ \mathtt{end}, \gamma_1 \cdot \gamma_2) = (\mathtt{local}\ \mathit{spec}_1'\ \mathtt{in}\ \mathit{spec}_2'\ \mathtt{end}, \gamma_1' \cdot \gamma_2')$
$\quad \mathit{where}\ \mathrm{strip}(\mathit{spec}_i, \gamma_i) = (\mathit{spec}_i', \gamma_i'),\ i \in \{1, 2\}$
$\mathrm{strip}(\mathit{spec}_1\ \langle ; \rangle\ \mathit{spec}_2, \gamma_1 \cdot \gamma_2) = (\mathit{spec}_1'\ \langle ; \rangle\ \mathit{spec}_2', \gamma_1' \cdot \gamma_2')$
$\quad \mathit{where}\ \mathrm{strip}(\mathit{spec}_i, \gamma_i) = (\mathit{spec}_i', \gamma_i'),\ i \in \{1, 2\}$
$\mathrm{strip}(\mathit{spec}, \gamma) = (\mathit{spec}, \gamma)$ otherwise

5.8 Type Explication

A signature $(N)S$ is *type-explicit* if, whenever $t \in N$ and t occurs free in S, then some substructure of S contains a type environment TE such that $TE(\mathit{tycon}) = (t, CE)$ for some *tycon* and some CE.

5.9 Signature Instantiation

A structure S_2 *is an instance of* a signature $\Sigma_1 = (N_1)S_1$, written $\Sigma_1 \geq S_2$, if there exists a realisation φ such that $\varphi(S_1) = S_2$ and $\mathrm{Supp}\,\varphi \subseteq N_1$. We write $\Sigma_1 \geq_\varphi S_2$ if we want to make φ explicit. (Note that if Σ_1 is type-explicit then there is at most one such φ.) A signature $\Sigma_2 = (N_2)S_2$ *is an instance of* $\Sigma_1 = (N_1)S_1$, written $\Sigma_1 \geq \Sigma_2$, if $\Sigma_1 \geq S_2$ and $N_2 \cap (\mathrm{names}\,\Sigma_1) = \emptyset$. It can be shown that $\Sigma_1 \geq \Sigma_2$ iff, for all S, whenever $\Sigma_2 \geq S$ then $\Sigma_1 \geq S$.

5.10 Functor Signature Instantiation

A pair $(S, (N')S')$ is called a *functor instance*. Given $\Phi = (N_1)(S_1, (N_1')S_1')$, a functor instance $(S_2, (N_2')S_2')$ is an *instance* of Φ, written $\Phi \geq (S_2, (N_2')S_2')$, if there exists a realisation φ such that $\varphi(S_1, (N_1')S_1') = (S_2, (N_2')S_2')$ and $\mathrm{Supp}\,\varphi \subseteq N_1$. Again we write $\Phi \geq_\varphi (S_2, (N_2')S_2')$ to make φ explicit.

5.11 Enrichment

In matching a structure to a signature, the structure will be allowed both to have more components, and to be more polymorphic, than (an instance of) the

signature. Precisely, we define enrichment of structures, environments and type structures by mutual recursion as follows.

A structure $S_1 = (m_1, E_1)$ *enriches* another structure $S_2 = (m_2, E_2)$, written $S_1 \succ S_2$, if

1. $m_1 = m_2$
2. $E_1 \succ E_2$

An environment E_1 *enriches* another environment E_2, $E_i = (SE_i, TE_i, VE_i)$, written $E_1 \succ E_2$, if

1. $\mathrm{Dom}\, SE_1 \supseteq \mathrm{Dom}\, SE_2$, and $\forall strid \in \mathrm{Dom}\, SE_2.\ SE_1(strid) \succ SE_2(strid)$
2. $\mathrm{Dom}\, TE_1 \supseteq \mathrm{Dom}\, TE_2$, and $\forall tycon \in \mathrm{Dom}\, TE_2.\ TE_1(tycon) \succ TE_2(tycon)$
3. $\mathrm{Dom}\, VE_1 \supseteq \mathrm{Dom}\, VE_2$, and $\forall id \in \mathrm{Dom}\, VE_2.\ VE_1(id) \succ VE_2(id)$

Finally, a type structure (θ_1, CE_1) *enriches* another type structure (θ_2, CE_2), written $(\theta_1, CE_1) \succ (\theta_2, CE_2)$, if

1. $\theta_1 = \theta_2$
2. Either $CE_1 = CE_2$ or $CE_2 = \{\}$

5.12 Signature Matching

A structure S *matches* a signature Σ_1 if there exists a structure S^- such that $\Sigma_1 \geq S^- \prec S$. Thus matching is a combination of instantiation and enrichment. There is at most one such S^-, given Σ_1 and S. Moreover, writing $\Sigma_1 = (N_1)S_1$, if $\Sigma_1 \geq S^-$ then there exists a realisation φ with $\mathrm{Supp}\, \varphi \subseteq N_1$ and $\varphi(S_1) = S^-$. We shall then say that S matches Σ_1 *via* φ. (Note that if Σ_1 is type-explicit then φ is uniquely determined by Σ_1 and S.)

A signature Σ_2 *matches* a signature Σ_1 if for all structures S, if S matches Σ_2 then S matches Σ_1. It can be shown that $\Sigma_2 = (N_2)S_2$ matches $\Sigma_1 = (N_1)S_1$ if and only if there exists a realisation φ with $\mathrm{Supp}\, \varphi \subseteq N_1$ and $\varphi(S_1) \prec S_2$ and $N_2 \cap \mathrm{names}\, \Sigma_1 = \emptyset$.

5.13 Principal Signatures

The definitions in this section concern the elaboration of signature expressions; more precisely they concern inferences of sentences of the form $B \vdash sigexp \Rightarrow S, \gamma$, where S is a structure and B is a basis. Recall, from Section 5.5, that the assembly of all semantic objects in such an inference must be admissible.

For any basis B and any structure S, we say that B *covers* S if for every substructure (m, E) of S such that $m \in N$ of B:

1. For every structure identifier $strid \in \mathrm{Dom}\, E$, B contains a substructure (m, E') with m free and $strid \in \mathrm{Dom}\, E'$

2. For every type constructor $tycon \in \mathrm{Dom}\, E$, B contains a substructure (m, E') with m free and $tycon \in \mathrm{Dom}\, E'$

(This condition is not a consequence of consistency of $\{B, S\}$; informally, it states that if S shares a substructure with B, then S mentions no more components of the substructure than B does.)

We say that a signature $(N)S$ with a trace γ is *principal for sigexp in B* if, choosing N so that $(N \text{ of } B) \cap N = \emptyset$,

1. B covers S

2. $B \vdash sigexp \Rightarrow S, \gamma$

3. Whenever $B \vdash sigexp \Rightarrow S', \gamma'$, then $(N)S \geq_\varphi S'$ and $\varphi'(\gamma) \succ \gamma'$, for some realisations φ and φ' such that $\mathrm{Supp}\, \varphi' \cap (N \text{ of } B) = \emptyset$ and such that φ' restricted to N is the same as φ.

We claim that if *sigexp* elaborates in B to some structure covered by B, then it possesses a principal signature in B (with some trace).[5]

Analogous to the definition given for type environments in Section 4.9, we say that a semantic object A *respects equality* if every type environment occurring in A respects equality.

Now let us assume that *sigexp* possesses a principal signature $\Sigma_0 = (N_0)S_0$ with γ_0 in B. We wish to define, in terms of Σ_0, another signature Σ with γ which provides more information about the equality attributes of structures which will match Σ_0. To this end, let T_0 be the set of type names $t \in N_0$ which do not admit equality, and such that (t, CE) occurs in S_0 for some $CE \neq \{\}$. Then we say Σ with γ is *equality-principal for sigexp in B* if

1. Σ respects equality

2. Σ and γ are obtained from Σ_0 and γ_0 just by making as many members of T_0 admit equality as possible, subject to 1. above.

It is easy to show that, if any such a pair (Σ, γ) exists, it is determined uniquely by (Σ_0, γ_0); moreover, Σ exists if Σ_0 itself respects equality.

We do not express equality-principality of signature elaboration by higher-order rules, in spite of comments in Section 1.2 which suggest this. The problem is that the obvious higher-order rule is on the one hand (slightly) incompatible with SML and on the other hand makes signature elaboration undecidable. One can repair the undecidability flaw by making a few rather innocent-looking changes to the semantics, but this would widen the gap between SML and EML.

[5] This claim must be slightly qualified, since it may be ill-formed in a mild sense. This is discussed at the end of Section 11.3 of [MT91].

5.14 Inference Rules

As for the Core, the rules of the Modules static semantics allow sentences of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

to be inferred, where in this case A is usually either a basis or a context and A' is a semantic object or an assembly of such objects.

Structure Expressions $\boxed{B \vdash \mathit{strexp} \Rightarrow S, \gamma}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E, \gamma \qquad m \notin (N \text{ of } B) \cup \text{names}\, E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (m, E), m \cdot \gamma} \tag{53}$$

$$\frac{B(\mathit{longstrid}) = S}{B \vdash \mathit{longstrid} \Rightarrow S, \epsilon} \tag{54}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow S, \gamma \\ B(\mathit{funid}) \geq_\varphi (S'', (N')S') \ ,\ S \succ S'' \qquad (N \text{ of } B) \cap N' = \emptyset \end{array}}{B \vdash \mathit{funid}\ (\ \mathit{strexp}\ ) \ \Rightarrow S', \varphi \cdot \gamma} \tag{55}$$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E, \gamma \qquad B \oplus E \vdash \mathit{strexp} \Rightarrow S, \gamma'}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow S, \gamma \cdot \gamma'} \tag{56}$$

Comments:

(53) The side condition ensures that each generative structure expression receives a new name. If the expression occurs in a functor body the structure name will be bound by (N') in rule 99; this will ensure that for each application of the functor, by rule 55, a new distinct name will be chosen for the structure generated.

(55) The side condition $(N \text{ of } B) \cap N' = \emptyset$ can always be satisfied by renaming bound names in $(N')S'$ thus ensuring that the generated structures receive new names.

Let $B(\mathit{funid}) = (N)(S_f, (N')S'_f)$. Assuming that $(N)S_f$ is type-explicit, the realisation φ for which $\varphi(S_f, (N')S'_f) = (S'', (N')S')$ is uniquely determined by S, since $S \succ S''$ can only hold if the type names and structure names in S and S'' agree. Recall that enrichment $\succ$ allows more components and more polymorphism, while instantiation $\geq$ does not.

Sharing between argument and result specified in the declaration of the functor *funid* is represented by the occurrence of the same name in both S_f and S'_f, and this repeated occurrence is preserved by φ, yielding sharing between the argument structure S and the result structure S' of this functor application.

(56) The use of $\oplus$, here and elsewhere, ensures that structure and type names generated by the first sub-phrase are distinct from names generated by the second sub-phrase.

Structure-level Declarations

$\boxed{B \vdash \mathit{strdec} \Rightarrow E, \gamma}$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E, \gamma \qquad N = \text{names}\,\gamma \setminus N \text{ of } B \qquad \frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E', \gamma'}{(N)\gamma \succ \gamma'}}{B \vdash \mathit{dec} \Rightarrow E, \gamma} \tag{57}$$

$$\frac{B \vdash \mathit{ax} \Rightarrow \gamma \qquad \frac{B \vdash \mathit{ax} \Rightarrow \gamma'}{\text{Clos}\gamma \succ \gamma'}}{B \vdash \texttt{axiom}\ \mathit{ax} \Rightarrow \{\} \text{ in Env}, \text{Clos}\gamma} \tag{57.1}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow SE, \gamma}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow SE \text{ in Env}, \gamma} \tag{58}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1, \gamma_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2} \tag{59}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}, \epsilon} \tag{60}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1, \gamma_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2} \tag{61}$$

Comments:

(57) The last premise can be seen as requiring *principality* of the trace γ (for *dec* in B), which implies principality of the environment E in the sense of the SML definition; it is a stronger condition, as γ may also include types which are not in E.

(57.1) The second premise ensures principality of the trace γ.

Axioms

$\boxed{B \vdash \mathit{ax} \Rightarrow \gamma}$

$$\frac{B \vdash \mathit{axexp} \Rightarrow \gamma \qquad \langle B \vdash \mathit{ax} \Rightarrow \gamma' \rangle}{B \vdash \mathit{axexp}\ \langle \texttt{and}\ \mathit{ax} \rangle \Rightarrow \gamma \langle \cdot \gamma' \rangle} \tag{61.1}$$

Comment: Axioms are *not* implicitly universally quantified over all their free variables. Such implicit quantification is convenient for presenting small examples, but the redundancy introduced by requiring variables to be explicitly quantified is helpful in detecting typographical errors in larger examples.

Axiomatic Expressions

$\boxed{B \vdash axexp \Rightarrow \gamma}$

$$\frac{C \text{ of } B \vdash exp^{\bullet} \Rightarrow \texttt{bool}, \emptyset, \gamma}{B \vdash exp^{\bullet} \Rightarrow \gamma} \tag{61.2}$$

Comment: Axiomatic expressions must be of type `bool` and are not allowed to contain unguarded explicit type variables.

Structure Bindings

$\boxed{B \vdash strbind \Rightarrow SE, \gamma}$

$$\frac{B \vdash sglstrbind \Rightarrow SE, \gamma \qquad \langle B + \text{names}\, SE \vdash strbind \Rightarrow SE', \gamma' \rangle}{B \vdash sglstrbind\ \langle \texttt{and}\ strbind \rangle \Rightarrow SE\ \langle +SE' \rangle, \gamma \langle \cdot \gamma' \rangle} \tag{62}$$

Single Structure Bindings

$\boxed{B \vdash sglstrbind \Rightarrow SE, \gamma}$

$$\frac{\begin{array}{c} B \vdash psigexp \Rightarrow (N)S, \gamma \qquad B \vdash strexp \Rightarrow S', \gamma' \\ N \cap N \text{ of } B = \emptyset \qquad (N)S \geq S'' \prec S' \end{array}}{B \vdash strid : psigexp\ \texttt{=}\ strexp \Rightarrow \{strid \mapsto S\},\ \gamma \cdot \gamma'} \tag{62.1}$$

$$\frac{B \vdash psigexp \Rightarrow (N)S, \gamma \qquad N \cap N \text{ of } B = \emptyset}{B \vdash strid : psigexp\ \texttt{=}\ \texttt{?} \Rightarrow \{strid \mapsto S\},\ \gamma} \tag{62.2}$$

$$\frac{B \vdash strexp \Rightarrow S, \gamma}{B \vdash strid\ \texttt{=}\ strexp \Rightarrow \{strid \mapsto S\},\ \gamma} \tag{62.3}$$

Comments:

(62.1) In EML, structures are like *abstractions* [MacQ86]: the signature of a structure is taken to be exactly the explicit signature. Any additional sharing present in the structure body is invisible outside the body. The type/structure names in S' are not accessible outside the body and so they may be safely reused (in contrast to SML, where *strid* is bound to S'').

(62.1),(62.2) The only difference between these two rules is that in the case where a body is present, it is required to elaborate and to fit the signature given. However, this does not effect the overall result of elaboration, which depends only on the signature given.

The side-condition $N \cap N$ of $B = \emptyset$ can always be satisfied by an appropriate α-conversion of $(N)S$.

Signature Expressions

$\boxed{B \vdash \mathit{sigexp} \Rightarrow S, \gamma}$

$$\frac{B \vdash \mathit{spec} \Rightarrow E, \gamma}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E), m \cdot \gamma} \tag{63}$$

$$\frac{B(\mathit{sigid}) \geq_{\varphi} S}{B \vdash \mathit{sigid} \Rightarrow S, \varphi} \tag{64}$$

Comments:

(63) In contrast to rule 53, m is not here required to be new. The name m may be chosen to achieve the sharing required in rule 88, or to achieve the enrichment side conditions of rule 62.1 or 99. The choice of m must result in an admissible object.

(64) The instance S of $B(\mathit{sigid})$ is not determined by this rule, but — as in rule 63 — the instance may be chosen to achieve sharing properties or enrichment conditions.

Principal Signatures

$\boxed{B \vdash \mathit{psigexp} \Rightarrow \Sigma, \gamma}$

$$\frac{\begin{array}{c}(N)S \text{ with } \gamma \text{ equality-principal for } \mathit{sigexp} \text{ in } B \qquad (N)S \text{ type-explicit} \\ \mathrm{strip}(\mathit{sigexp}, \gamma) = (\mathit{sigexp}', \gamma') \\ (N)S \text{ with } \gamma' \text{ equality-principal for } \mathit{sigexp}' \text{ in } B\end{array}}{B \vdash \mathit{sigexp} \Rightarrow (N)S, (N)\gamma} \tag{65}$$

Comment: $B \vdash \mathit{sigexp} \Rightarrow S, \gamma$ follows from the definition of equality-principality of $(N)S$ with γ for *sigexp* in B. The purpose of the second equality-principality requirement is to ensure that dropping axioms from *sigexp* does not change the result of elaboration. We also need principality of the second elaboration to get a unique trace γ. In an implementation, the two equality-principality requirements will probably correspond to two passes of static analysis: the first pass elaborates the signature expression stripped of its axioms; the second pass then includes the axioms and checks that no further identification of type names is required for elaboration to succeed.

Signature Declarations

$\boxed{B \vdash \mathit{sigdec} \Rightarrow G, \gamma}$

$$\frac{B \vdash \mathit{sigbind} \Rightarrow G, \gamma}{B \vdash \texttt{signature}\ \mathit{sigbind} \Rightarrow G, \gamma} \tag{66}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}, \epsilon} \tag{67}$$

$$\frac{B \vdash \mathit{sigdec}_1 \Rightarrow G_1, \gamma_1 \qquad B + G_1 \vdash \mathit{sigdec}_2 \Rightarrow G_2, \gamma_2}{B \vdash \mathit{sigdec}_1\ \langle ; \rangle\ \mathit{sigdec}_2 \Rightarrow G_1 + G_2, \gamma_1 \cdot \gamma_2} \tag{68}$$

Comments:

(66) The first closure restriction of Section 3.6 can be enforced by replacing the B in the premise by $(B \cap B_0) + G$ of B.

(68) A signature declaration does not create any new structures or types; hence the use of $+$ instead of $\oplus$.

Signature Bindings

$\boxed{B \vdash \mathit{sigbind} \Rightarrow G, \gamma}$

$$\frac{B \vdash \mathit{psigexp} \Rightarrow \Sigma, \gamma \qquad \langle B \vdash \mathit{sigbind} \Rightarrow G, \gamma' \rangle}{B \vdash \mathit{sigid}\ \texttt{=}\ \mathit{psigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \Rightarrow \{\mathit{sigid} \mapsto \Sigma\}\ \langle + G \rangle,\ \gamma \langle \cdot \gamma' \rangle} \tag{69}$$

Comment: The condition that Σ be equality-principal, implicit in the first premise, ensures that the signature found is as general as possible given the sharing constraints present in *psigexp*.

Specifications

$\boxed{B \vdash \mathit{spec} \Rightarrow E, \gamma}$

$$\frac{C \text{ of } B \vdash \mathit{valdesc} \Rightarrow VE}{B \vdash \texttt{val}\ \mathit{valdesc} \Rightarrow \text{Clos}VE \text{ in Env},\ \text{Clos}VE \text{ in Trace}} \tag{70}$$

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE}{B \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow TE \text{ in Env},\ TE \text{ in Trace}} \tag{71}$$

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE \qquad \forall (\theta, CE) \in \text{Ran}\, TE,\ \theta \text{ admits equality}}{B \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow TE \text{ in Env},\ TE \text{ in Trace}} \tag{72}$$

$$\frac{C \text{ of } B + TE \vdash \mathit{datdesc} \Rightarrow VE, TE \qquad E = (\{\}, TE, VE)}{B \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow E,\ E \text{ in Trace}} \tag{73}$$

$$\frac{C \text{ of } B \vdash \mathit{exdesc} \Rightarrow VE \qquad E = (\{\}, \{\}, VE)}{B \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow E,\ E \text{ in Trace}} \tag{74}$$

$$\frac{B \vdash \mathit{axdesc} \Rightarrow \gamma}{B \vdash \texttt{axiom}\ \mathit{axdesc} \Rightarrow \{\} \text{ in Env},\ \gamma} \tag{74.1}$$

$$\frac{B \vdash \mathit{strdesc} \Rightarrow SE, \gamma}{B \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow SE \text{ in Env},\ \gamma} \tag{75}$$

$$\frac{B \vdash \mathit{shareq} \Rightarrow \{\}}{B \vdash \texttt{sharing}\ \mathit{shareq} \Rightarrow \{\} \text{ in Env},\ \epsilon} \tag{76}$$

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow E_1, \gamma_1 \qquad B + E_1 \vdash \mathit{spec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \texttt{local}\ \mathit{spec}_1\ \texttt{in}\ \mathit{spec}_2\ \texttt{end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2} \tag{77}$$

$$\frac{B(longstrid_1) = (m_1, E_1) \quad \cdots \quad B(longstrid_n) = (m_n, E_n)}{B \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n,\ \epsilon} \tag{78}$$

$$\frac{B(sigid_1) \geq_{\varphi_1} (m_1, E_1) \quad \cdots \quad B(sigid_n) \geq_{\varphi_n} (m_n, E_n)}{B \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow E_1 + \cdots + E_n,\ \varphi_1 \cdot \ldots \cdot \varphi_n} \tag{79}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env},\ \epsilon} \tag{80}$$

$$\frac{B \vdash spec_1 \Rightarrow E_1, \gamma_1 \qquad B + E_1 \vdash spec_2 \Rightarrow E_2, \gamma_2}{B \vdash spec_1\ \langle ; \rangle\ spec_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2} \tag{81}$$

Comments:

(70) VE is determined by B and *valdesc*.

(71)–(73) The type functions in TE may be chosen to achieve the sharing hypothesis of rule 89 or the enrichment conditions of rules 62.1 and 99. In particular, the type names in TE in rule 73 need not be new. Also, in rule 71 the type functions in TE may admit equality.

(74) VE is determined by B and *exdesc* and contains monotypes only.

(79) The names m_i in the instances may be chosen to achieve sharing or enrichment conditions.

Value Descriptions

$\boxed{C \vdash valdesc \Rightarrow VE}$

$$\frac{var = id^{\mathtt{V}} \qquad C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE \rangle}{C \vdash var : ty\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{id \mapsto \tau\}\ \langle + VE \rangle} \tag{82}$$

Type Descriptions

$\boxed{C \vdash typdesc \Rightarrow TE}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad \langle C \vdash typdesc \Rightarrow TE \rangle \qquad \text{arity}\,\theta = k}{C \vdash tyvarseq\ tycon\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \{tycon \mapsto (\theta, \{\})\}\ \langle + TE \rangle} \tag{83}$$

Comment: Note that any θ of arity k may be chosen but that the constructor environment in the resulting type structure must be empty. For example,

```
datatype s=c
type t
sharing type s=t
```

is a legal specification, but the type structure bound to `t` does not bind any value constructors.

Datatype Descriptions $\boxed{C \vdash datdesc \Rightarrow VE, TE}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)} t \vdash condesc \Rightarrow CE \qquad \langle C \vdash datdesc \Rightarrow VE, TE \rangle}{\begin{array}{c} C \vdash tyvarseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \mathrm{Clos}CE\langle + VE \rangle,\ \{tycon \mapsto (t, \mathrm{Clos}CE)\}\ \langle + TE \rangle \end{array}} \tag{84}$$

Constructor Descriptions $\boxed{C, \tau \vdash condesc \Rightarrow CE}$

$$\frac{con = id^{\mathsf{C}} \qquad \langle C, \tau \vdash condesc \Rightarrow CE \rangle}{C, \tau \vdash con\ \langle\ \texttt{|}\ condesc \rangle \Rightarrow \{id \mapsto \tau\}\ \langle + CE \rangle} \tag{85}$$

$$\frac{con = id^{\mathsf{C}} \qquad C \vdash ty \Rightarrow \tau' \qquad \langle C, \tau \vdash condesc \Rightarrow CE \rangle}{C, \tau \vdash con\ \texttt{of}\ ty\ \langle\ \texttt{|}\ condesc \rangle \Rightarrow \{id \mapsto \tau' \to \tau\}\ \langle + CE \rangle} \tag{85.1}$$

Exception Descriptions $\boxed{C \vdash exdesc \Rightarrow VE}$

$$\frac{excon = id^{\mathsf{E}} \qquad \langle C \vdash exdesc \Rightarrow VE \rangle}{C \vdash excon\ \langle \texttt{and}\ exdesc \rangle \Rightarrow \{id \mapsto \texttt{exn}\}\ \langle + VE \rangle} \tag{86}$$

$$\frac{\begin{array}{c} C \vdash ty \Rightarrow \tau \qquad \mathrm{tyvars}(ty) = \emptyset \\ excon = id^{\mathsf{E}} \qquad \langle C \vdash exdesc \Rightarrow VE \rangle \end{array}}{C \vdash excon\ \texttt{of}\ ty\ \langle \texttt{and}\ exdesc \rangle \Rightarrow \{id \mapsto \tau \to \texttt{exn}\} \langle + VE \rangle} \tag{86.1}$$

Comments:

(86.1) The requirement that there are no type variables in ty (rather than in τ, as in rule 31.1 or in rule 86 of [MTH90]) was suggested in Appendix D in [MT91]. The problem with the requirement in [MTH90] is that principality is lost because of the existence of type functions like $\Lambda\alpha.\texttt{int}$; see [MT91]. Here is an example:

```
type 'a t
exception e of 'a t
```

This specification does not elaborate because of the side-condition in 86.1. The weaker side-condition $\mathrm{tyvars}(\tau) = \emptyset$ would still allow successful elaborations with e.g. $\texttt{t} \mapsto \Lambda\alpha.\texttt{int}$.

Axiom Descriptions $\boxed{B \vdash axdesc \Rightarrow \gamma}$

Axiom descriptions, specification expressions, etc. do not add components to the basis, hence the only outcome of elaboration is the trace.

$$\frac{B \vdash specexp \Rightarrow \gamma \qquad \langle B \vdash axdesc \Rightarrow \gamma' \rangle}{B \vdash specexp\ \langle \texttt{and}\ axdesc \rangle \Rightarrow \gamma \langle \cdot \gamma' \rangle} \tag{86.2}$$

Specification Expressions

$\boxed{B \vdash \mathit{specexp} \Rightarrow \gamma}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E, \gamma \qquad B \oplus E \vdash \mathit{axexp} \Rightarrow \gamma'}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{axexp}\ \texttt{end} \Rightarrow \gamma \cdot \gamma'} \tag{86.3}$$

Structure Descriptions

$\boxed{B \vdash \mathit{strdesc} \Rightarrow \mathit{SE}, \gamma}$

$$\frac{B \vdash \mathit{sigexp} \Rightarrow S, \gamma \qquad \langle B \vdash \mathit{strdesc} \Rightarrow \mathit{SE}, \gamma' \rangle}{B \vdash \mathit{strid}\ :\ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{strdesc} \rangle \Rightarrow \{\mathit{strid} \mapsto S\}\ \langle + \mathit{SE} \rangle,\ \gamma \langle \cdot \gamma' \rangle} \tag{87}$$

Sharing Equations

$\boxed{B \vdash \mathit{shareq} \Rightarrow \{\}}$

$$\frac{m \text{ of } B(\mathit{longstrid}_1) = \cdots = m \text{ of } B(\mathit{longstrid}_n)}{B \vdash \mathit{longstrid}_1\ \texttt{=}\ \cdots\ \texttt{=}\ \mathit{longstrid}_n\ \Rightarrow \{\}} \tag{88}$$

$$\frac{\theta \text{ of } B(\mathit{longtycon}_1) = \cdots = \theta \text{ of } B(\mathit{longtycon}_n)}{B \vdash \texttt{type}\ \mathit{longtycon}_1\ \texttt{=}\ \cdots\ \texttt{=}\ \mathit{longtycon}_n\ \Rightarrow \{\}} \tag{89}$$

$$\frac{B \vdash \mathit{shareq}_1 \Rightarrow \{\} \qquad B \vdash \mathit{shareq}_2 \Rightarrow \{\}}{B \vdash \mathit{shareq}_1\ \texttt{and}\ \mathit{shareq}_2 \Rightarrow \{\}} \tag{90}$$

Comments:

(88) The premise is weaker than $B(\mathit{longstrid}_1) = \cdots = B(\mathit{longstrid}_n)$. Two different structures with the same name may be thought of as representing different views. The requirement that B is consistent forces different views to be consistent.

(89) The premise is weaker than $B(\mathit{longtycon}_1) = \cdots = B(\mathit{longtycon}_n)$. A type structure with empty constructor environment may have the same type name as one with a non-empty constructor environment; the former could arise from a type description, and the latter from a datatype description. However, the requirement that B is consistent will prevent two type structures with constructor environments which have different non-empty domains from sharing the same type name.

Functor Specifications etc.

Several rules have been removed here, because EML does not support functor specifications, functor descriptions or functor signature expressions. In SML these constructs are assigned semantics although they cannot appear in programs.

Deleted (91)

Deleted (92)

Deleted (93)

Deleted (94)

Deleted (95)

Functor Declarations $\boxed{B \vdash \mathit{fundec} \Rightarrow F, \gamma}$

$$\frac{B \vdash \mathit{funbind} \Rightarrow F, \gamma}{B \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow F, \gamma} \tag{96}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}, \epsilon} \tag{97}$$

$$\frac{B \vdash \mathit{fundec}_1 \Rightarrow F_1, \gamma_1 \qquad B + F_1 \vdash \mathit{fundec}_2 \Rightarrow F_2, \gamma_2}{B \vdash \mathit{fundec}_1\ \langle ; \rangle\ \mathit{fundec}_2 \Rightarrow F_1 + F_2, \gamma_1 \cdot \gamma_2} \tag{98}$$

Comments:

(96) The second closure restriction of Section 3.6 can be enforced by replacing the B in the premise by $(B \cap B_0) + (G \text{ of } B) + (F \text{ of } B)$.

Functor Bindings $\boxed{B \vdash \mathit{funbind} \Rightarrow F, \gamma}$

$$\frac{\begin{array}{c} B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma_1 \qquad N \cap N \text{ of } B = \emptyset \qquad B' = B \oplus \{\mathit{strid} \mapsto S\} \\ B' \vdash \mathit{psigexp}' \Rightarrow \Sigma, \gamma_2 \qquad B' \vdash \mathit{strexp} \Rightarrow S', \gamma_3 \\ \Sigma \geq S'' \prec S' \qquad \langle B \vdash \mathit{funbind} \Rightarrow F, \gamma_4 \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ (\ \mathit{strid} : \mathit{psigexp}\ )\ [:\ \mathit{psigexp}'\ \texttt{=}\ \mathit{strexp}]\ \langle \texttt{and}\ \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, \Sigma)\}\ \langle + F \rangle,\ \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \langle \cdot \gamma_4 \rangle \end{array}} \tag{99}$$

$$\frac{\begin{array}{c} B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma_1 \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{psigexp}' \Rightarrow \Sigma, \gamma_2 \\ N \cap N \text{ of } B = \emptyset \qquad \langle B \vdash \mathit{funbind} \Rightarrow F, \gamma_3 \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ (\ \mathit{strid} : \mathit{psigexp}\ )\ :\ \mathit{psigexp}'\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, \Sigma)\}\ \langle + F \rangle,\ \gamma_1 \cdot \gamma_2 \langle \cdot \gamma_3 \rangle \end{array}} \tag{99.1}$$

Comments:

(99) In EML, functors are like parameterised abstractions [MacQ86]: the output signature of a functor is taken to be exactly the explicit output signature. Any additional sharing present in the functor body is invisible outside the body. Compare rule 99 of [MTH90].

(99),(99.1) The requirement that $(N)S$ be equality-principal, implicit in the first premise, forces $(N)S$ to be as general as possible given the sharing constraints in *psigexp*. The requirement that $(N)S$ be type-explicit ensures that there is at most one realisation via which an actual argument can match $(N)S$. Since $\oplus$ is used, any structure name m and type name t in S acts like a constant in the functor body and in the functor result signature; in particular, it ensures that further names generated during elaboration of the body are distinct from m and t. The only difference between these two rules is that in the case where a body is present, it is required to elaborate and to fit the output signature. This does not affect the overall result of elaboration, which depends only on the input and output signatures.

Top-level Declarations

$\boxed{B \vdash \mathit{topdec} \Rightarrow B', \gamma}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E, \gamma}{B \vdash \mathit{strdec} \Rightarrow (\mathrm{names}\, E, E) \text{ in Basis, } \gamma} \quad (100)$$

$$\frac{B \vdash \mathit{sigdec} \Rightarrow G, \gamma}{B \vdash \mathit{sigdec} \Rightarrow (\mathrm{names}\, G, G) \text{ in Basis, } \gamma} \quad (101)$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F, \gamma}{B \vdash \mathit{fundec} \Rightarrow (\mathrm{names}\, F, F) \text{ in Basis, } \gamma} \quad (102)$$

6 Dynamic Semantics for the Core

6.1 Reduced Syntax

Since types are fully dealt with in the static semantics, the dynamic semantics ignores them. The Core syntax is therefore reduced by the following transformations, for the purpose of the dynamic semantics:

- All explicit type ascriptions "`:` *ty*" are omitted, and qualifications "`of` *ty*" are omitted from constructor and exception bindings.
- Any declaration of the form "`type` *typbind*"or "`eqtype` *typbind*" is replaced by the empty declaration.
- The Core phrase classes TypBind, Ty and TyRow are omitted.

6.2 Simple Objects

All objects in the dynamic semantics are built from identifier classes together with the simple object classes shown (with the variables which range over them) in Figure 14.

en	$\in$	ExName	exception names
b	$\in$	BasVal	basic values
sv	$\in$	SVal	special values
sp	$\in$	Bit $= \{\bot, \top\}$	flags
		{FAIL}	failure

Figure 14: Simple Semantic Objects

ExName is an infinite set; it is totally ordered, i.e. every subset A of ExName has a smallest element $\min A$. BasVal is described below in Section 6.4. SVal is the class of values denoted by the special constants SCon. Each integer or real constant denotes a value according to usual conventions for decimal numbers with limited precision; each string constant denotes a sequence of characters as explained in Section 2.2. The value denoted by *scon* is written val(*scon*). The values $\bot$ and $\top$ are only used as Boolean flags. If a semantic object x contains a Bit component, we write $x_\bot$ to denote the same object, but with the Bit component set to $\bot$. On Bit, we define the operation $\wedge$ as the greatest lower bound of the order $\bot \leq \top$; its generalisation to finite index sets I is written $\bigwedge_{i \in I}$. FAIL is the result of a failing attempt to match a value and a pattern. Thus FAIL is neither a value nor an exception, but simply a semantic object used in the rules to express operationally how matching proceeds.

$$
\begin{array}{rcl}
v & \in & \mathrm{Val} = \mathrm{SVal} \uplus \mathrm{BasVal} \uplus \mathrm{Con} \\
 & & \quad \uplus (\mathrm{Con} \times \mathrm{Val}) \uplus \mathrm{ExVal} \\
 & & \quad \uplus \mathrm{Record} \uplus \mathrm{Closure} \uplus \{\texttt{Incomplete}\} \\
r & \in & \mathrm{Record} = \mathrm{Lab} \xrightarrow{\text{fin}} \mathrm{Val} \\
e & \in & \mathrm{ExVal} = \mathrm{ExName} \uplus (\mathrm{ExName} \times \mathrm{Val}) \\
[e] \text{ or } p & \in & \mathrm{Pack} = \mathrm{ExVal} \\
(match, E, VE) & \in & \mathrm{Closure} = \mathrm{Match} \times \mathrm{Env} \times \mathrm{VarEnv} \\
ens & \in & \mathrm{ExNameSet} = \mathrm{Fin}(\mathrm{ExName}) \\
(sp, ens) \text{ or } s & \in & \mathrm{State} = \mathrm{Bit} \times \mathrm{ExNameSet} \\
(SE, VE) \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{VarEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\text{fin}} \mathrm{Env} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Id} \xrightarrow{\text{fin}} \mathrm{Val}
\end{array}
$$

Figure 15: Compound Semantic Objects

Exception constructors evaluate to exception names, unlike value constructors which simply evaluate to themselves. This is to accommodate the generative nature of exception bindings; each evaluation of a declaration of an exception constructor binds it to a new unique name.

6.3 Compound Objects

The compound objects for the dynamic semantics are shown in Figure 15. Many conventions and notations are adopted as in the static semantics; in particular projection, injection and modification all retain their meaning. The value `Incomplete` is used to represent the "value" associated with a variable having an undefined value (e.g. because it was bound using "`?`").

We take $\uplus$ to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint. A particular case deserves mention; ExVal and Pack (exception values and packets) are isomorphic classes, but the latter class corresponds to exceptions which have been raised, and therefore has different semantic significance from the former, which is just a subclass of values.

Although the same names, e.g. E for an environment, are used as in the static semantics, the objects denoted are different. This need cause no confusion since the static and dynamic semantics are presented separately. An important point is that structure names m have no significance at all in the dynamic semantics; this explains why the object class $\mathrm{Str} = \mathrm{StrName} \times \mathrm{Env}$ is absent here – for the dynamic semantics the concepts *structure* and *environment* coincide.

6.4 Basic Values

The basic values in BasVal are the values bound to predefined variables. These values are denoted by the identifiers to which they are bound in the initial basis (see Appendix C), and are as follows:

```
abs floor real sqrt sin cos arctan exp ln
  size chr ord explode implode div mod
        / * + - = <> < > <= >=
```

The meaning of basic values (all of which are functions) is represented by the function

$$\text{APPLY} \ : \ \text{BasVal} \times \text{Val} \rightarrow \text{Val} \uplus \text{Pack}$$

which is detailed in Appendix C.

6.5 Basic Exceptions

A subset BasExName $\subset$ ExName of the exception names are bound to predefined exception constructors. These names are denoted by the identifiers to which all but the last are bound in the initial basis (see Appendix C), and are as follows:

```
Abs  Ord  Chr   Div  Mod  Quot  Prod
Neg  Sum  Diff  Floor  Sqrt  Exp  Ln
Match  Bind  Interrupt  NoCode  Abuse
```

The exceptions on the first two lines are raised by corresponding basic functions, where `~ / * + -` correspond respectively to `Neg Quot Prod Sum Diff`. The details are given in Appendix C. The exceptions `Match` and `Bind` are raised upon failure of pattern-matching in evaluating a function `fn` *match* or a *valbind*, as detailed in the rules to follow. `Interrupt` is raised by external intervention. The exception `NoCode` is raised to signal an attempt to evaluate a specification construct, see below. Finally, `Abuse` is raised in the verification semantics only, to signal abuse of the convergence predicate (rule 216).

Recall from Section 4.11 that in the context `fn` *match*, the *match* must be irredundant and exhaustive and that the compiler should flag the *match* if it violates these restrictions. The exception `Match` can only be raised for a match which is not exhaustive, and has therefore been flagged by the compiler.

For each value binding *pat* `=` *exp* the compiler must issue a report (but still compile) if *either* *pat* is not exhaustive *or* *pat* contains no variable. This will (on both counts) detect a mistaken declaration like `val nil =` *exp* in which the user expects to declare a new variable `nil` (whereas the language dictates that `nil` is here a constant pattern, so no variable gets declared). However, these warnings should not be given when the binding is a component of a top-level declaration `val` *valbind*; e.g. `val x::l =` exp_1 `and y =` exp_2 is not faulted by the compiler at top level, but may of course generate a `Bind` exception.

The exception `NoCode` is raised when evaluation encounters an expression having an undefined value (i.e. an atomic expression of the form `?`, an expression of the form "`exists` $match^\bullet$", "`forall` $match^\bullet$", "$exp^\bullet$ `terminates`", "$exp_1^\bullet$ `==` $exp_2^\bullet$", or accessing a variable which has been bound to `Incomplete`). This exception is *not* bound in the initial basis (but `NoCode`, `Abuse` $\in$ BasExName to avoid erroneous reuse of these names for user-declared exceptions). Rule 121.1 and premises on rules 120 and 121 guarantee that the exception `NoCode` cannot be caught by an explicit handler in the program. This is required to ensure that replacing "`?`" by code will not change the result of an evaluation, except from [`NoCode`] to something else, provided the evaluation of the new code yields a value. The exception `NoCode` is handled as a special case in one of the rules for value binding (rule 135.1) — if evaluation of the expression on the right-hand side of a value binding raises `NoCode`, then the binding is done with the value `Incomplete`.

6.6 Closures

The informal understanding of a *closure* $(match, E, VE)$ is as follows: when the closure is applied to a value v, *match* will be evaluated against v, in the environment E modified in a special sense by VE. The domain $\mathrm{Dom}\, VE$ of this third component contains those function identifiers to be treated recursively in the evaluation. To achieve this effect, the evaluation of *match* will take place not in $E + VE$ but in $E + \mathrm{Rec}\, VE$, where

$$\mathrm{Rec} \;:\; \mathrm{VarEnv} \to \mathrm{VarEnv$$

is defined as follows:

- $\mathrm{Dom}(\mathrm{Rec}\, VE) = \mathrm{Dom}\, VE$
- If $VE(id) \notin \mathrm{Closure}$, then $(\mathrm{Rec}\, VE)(id) = VE(id)$
- If $VE(id) = (match', E', VE')$ then $(\mathrm{Rec}\, VE)(id) = (match', E', VE)$

The effect is that, before application of $(match, E, VE)$ to v, the closure values in $\mathrm{Ran}\, VE$ are "unrolled" once, to prepare for their possible recursive application during the evaluation of *match* upon v.

This device is adopted to ensure that all semantic objects are finite (by controlling the unrolling of recursion). The operator Rec is invoked in just two places in the semantic rules: in the rule for recursive value bindings of the form "`rec` *valbind*", and in the rule for evaluating an application expression "*exp atexp*" in the case that *exp* evaluates to a closure.

States and Flags

A *state* consists of a set of exception names and a flag. The set of exception names records the exceptions introduced so far. The flag indicates whether or not a *specification construct* (e.g. a quantified expression) has been encountered during the

evaluation so far; this includes specification constructs within values, see the next section. Initially, this component is $\top$ and each specification construct makes it $\bot$. The meaning of specification constructs in dynamic and verification semantics differs in a significant way. The purpose of the flag is to relate dynamic and verification semantics, see Section 8.7 and the rules for the convergence predicate in the verification semantics, rules 215 to 217. It is also used in the definition of the meaning of quantification in the verification semantics (rules 211–214) where we want to quantify over ML-definable values only.

Let $s = (sp, ens)$. We define the notation $s \wedge sp'$ as abbreviation for the state $(sp \wedge sp', ens)$.

In contrast to SML, states do not contain a map from "addresses" to values.

Pure Values

We define a function AI : Val $\to$ Bit to indicate whether or not a value is *pure*, meaning that it does not depend on specification constructs. For example, the expression `(fn y=>y terminates)` evaluates to an impure value. Impure values can only arise if functional types are involved, since specification constructs in values can occur only within closures. Therefore, the function AI is used directly only in rules 123 (where closures are formed) and 104 (where closures may be extracted from an environment).

$\mathrm{AI}(v) = \top$ means that the value v neither directly nor indirectly depends on specification constructs, $\mathrm{AI}(v) = \bot$ means that there *may* be such a dependency. Because values include closures, we need corresponding auxiliary functions for various syntactic phrases; for simplicity, we call them all AI as well. The connection is (for expressions): if $s, E \vdash exp \Rightarrow v, s'$ is a derivable sentence in the dynamic semantics then $\mathrm{AI}(E, exp) \wedge sp \text{ of } s \leq \mathrm{AI}(v) \wedge sp \text{ of } s'$. The functions AI can be seen as an abstract interpretation, very much in the style of abstract interpretation for strictness analysis.

Another auxiliary function for AI is $\widehat{\mathrm{Rec}}$: VarEnv $\to$ VarEnv, defined as follows:

- $\mathrm{Dom}(\widehat{\mathrm{Rec}}\, VE) = \mathrm{Dom}\, VE$
- If $VE(id) \notin$ Closure, then $(\widehat{\mathrm{Rec}}\, VE)(id) = VE(id)$
- If $VE(id) = (match', E', VE')$ then

$$(\widehat{\mathrm{Rec}}\, VE)(id) = (match', E', VE + \{id \mapsto 1\}).$$

The operation $\widehat{\mathrm{Rec}}$ is very similar to Rec except that each identifier id in the variable environment of a closure will be "unrolled" at most once, which is achieved by binding id after one unrolling to the value 1, disregarding its type.

On values, we have $\mathrm{AI}(v) = \bot$ if v either contains `Incomplete` or is a closure that contains specification constructs or refers to incompletely defined identifiers:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\mathrm{AI} : \mathrm{Val} \to \mathrm{Bit}} \\
\mathrm{AI}(\texttt{Incomplete}) & = & \bot & \\
\mathrm{AI}(v) & = & \top & \text{if } v \in \mathrm{SVal} \uplus \mathrm{BasVal} \uplus \mathrm{Con} \uplus \mathrm{ExName} \\
\mathrm{AI}(con, v) & = & \mathrm{AI}(v) & \\
\mathrm{AI}(en, v) & = & \mathrm{AI}(v) & \\
\mathrm{AI}(r) & = & \bigwedge_{lab \in \mathrm{Dom}\, r} \mathrm{AI}(r(lab)) & \\
\mathrm{AI}(match, E, VE) & = & \mathrm{AI}(E + \widehat{\mathrm{Rec}}\, VE, match) &
\end{array}
$$

The function AI on the right-hand side of the last equation has the form $\mathrm{AI} : \mathrm{Env} \times \mathrm{Match} \to \mathrm{Bit}$. Similarly, we have a function $\mathrm{AI} : \mathrm{Env} \times \mathrm{Exp} \to \mathrm{Bit}$ for the phrase class Exp, etc. We adopt the notation of options in syntactic phrases for the definition of functions by equations: the option is either present in both sides of an equation or in neither.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{Env} \times \mathrm{Match} \to \mathrm{Bit}} \\
\mathrm{AI}(E, mrule\ \langle\ \texttt{|}\ match \rangle) & = & \mathrm{AI}(E, mrule)\ \langle \wedge\, \mathrm{AI}(E, match) \rangle \\
 & & \\
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{Env} \times \mathrm{Mrule} \to \mathrm{Bit}} \\
\mathrm{AI}(E, pat\ \texttt{=>}\ exp) & = & \mathrm{AI}(E + \mathrm{AI}(pat), exp)
\end{array}
$$

The abstract interpretation of a pattern (defined further below) is a variable environment that maps each *var* in *pat* to the dummy value 1. It does not depend on the environment because identifiers in patterns of the Bare language already come equipped with their status, i.e. a dynamic environment is neither necessary nor helpful to distinguish value variables from value constructors and exception constructors.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{Env} \times \mathrm{Exp} \to \mathrm{Bit}} \\
\mathrm{AI}(E, atexp) & = & \mathrm{AI}(E, atexp) \\
\mathrm{AI}(E, exp\ atexp) & = & \mathrm{AI}(E, exp) \wedge \mathrm{AI}(E, atexp) \\
\mathrm{AI}(E, exp_1^\bullet\ \texttt{==}\ exp_2^\bullet) & = & \bot \\
\mathrm{AI}(E, \texttt{exists}\ match^\bullet) & = & \bot \\
\mathrm{AI}(E, \texttt{forall}\ match^\bullet) & = & \bot \\
\mathrm{AI}(E, exp^\bullet\ \texttt{terminates}) & = & \bot \\
\mathrm{AI}(E, exp\ \texttt{handle}\ match) & = & \mathrm{AI}(E, exp) \wedge \mathrm{AI}(E, match) \\
\mathrm{AI}(E, \texttt{raise}\ exp) & = & \mathrm{AI}(E, exp) \\
\mathrm{AI}(E, \texttt{fn}\ match) & = & \mathrm{AI}(E, match)
\end{array}
$$

In the first equation, the AI on the right-hand side refers to the corresponding function for atomic expressions. Equality, quantification, and the convergence predicate are considered specification constructs.

$$
\begin{array}{lcl}
\mathrm{AI} : \mathrm{Env} \times \mathrm{AtExp} \to \mathrm{Bit} & & \\
\mathrm{AI}(E, \mathit{scon}) & = & \top \\
\mathrm{AI}(E, \mathit{longvar}) & = & \mathrm{AI}(E(\mathit{longvar})) \\
\mathrm{AI}(E, \mathit{longcon}) & = & \mathrm{AI}(E(\mathit{longcon})) \\
\mathrm{AI}(E, \mathit{longexcon}) & = & \mathrm{AI}(E(\mathit{longexcon})) \\
\mathrm{AI}(E, \{\ \langle \mathit{exprow} \rangle\ \}) & = & \top\ \langle \wedge\, \mathrm{AI}(E, \mathit{exprow}) \rangle \\
\mathrm{AI}(E, \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end}) & = & \mathit{sp} \wedge \mathrm{AI}(E + E', \mathit{exp}) \\
 & & \text{where } (\mathit{sp}, E') = \mathrm{AI}(E, \mathit{dec}) \\
\mathrm{AI}(E, \texttt{(}\ \mathit{exp}\ \texttt{)}) & = & \mathrm{AI}(E, \mathit{exp}) \\
\mathrm{AI}(E, \texttt{?}) & = & \bot
\end{array}
$$

The abstract interpretation of a `let`-expression `let` *dec* `in` *exp* `end` is $\bot$ if *dec* depends on specification constructs, even if the corresponding identifier is not used in *exp*. This reflects the call-by-value nature of evaluation.

Question marks are considered specification constructs when they are used as expressions.

$$
\begin{array}{l}
\mathrm{AI} : \mathrm{Env} \times \mathrm{ExpRow} \to \mathrm{Bit} \\
\mathrm{AI}(E, \mathit{lab}\ \texttt{=}\ \mathit{exp}\ \langle\ \texttt{,}\ \mathit{exprow} \rangle) \quad = \quad \mathrm{AI}(E, \mathit{exp})\ \langle \wedge\, \mathrm{AI}(E, \mathit{exprow}) \rangle
\end{array}
$$

The abstract interpretation of an (atomic) pattern is a variable environment mapping each value variable that occurs in the pattern to 1, regardless of its type. The value 1 is an arbitrary choice here; any value v with $\mathrm{AI}(v) = \top$ would do.

$$
\begin{array}{lcl}
\mathrm{AI} : \mathrm{Pat} \to \mathrm{VarEnv} & & \\
\mathrm{AI}(\mathit{atpat}) & = & \mathrm{AI}(\mathit{atpat}) \\
\mathrm{AI}(\mathit{longcon}\ \mathit{atpat}) & = & \mathrm{AI}(\mathit{atpat}) \\
\mathrm{AI}(\mathit{longexcon}\ \mathit{atpat}) & = & \mathrm{AI}(\mathit{atpat}) \\
\mathrm{AI}(\mathit{var}\ \texttt{as}\ \mathit{pat}) & = & \{\mathit{id} \mapsto 1\} + \mathrm{AI}(\mathit{pat}),\ \text{where } \mathit{var} = \mathit{id}^{\mathtt{V}} \\
 & & \\
\mathrm{AI} : \mathrm{AtPat} \to \mathrm{VarEnv} & & \\
\mathrm{AI}(_) & = & \{\} \\
\mathrm{AI}(\mathit{scon}) & = & \{\} \\
\mathrm{AI}(\mathit{var}) & = & \{\mathit{id} \mapsto 1\},\ \text{where } \mathit{var} = \mathit{id}^{\mathtt{V}} \\
\mathrm{AI}(\mathit{longcon}) & = & \{\} \\
\mathrm{AI}(\mathit{longexcon}) & = & \{\} \\
\mathrm{AI}(\{\ \langle \mathit{patrow} \rangle\ \}) & = & \{\}\ \langle + \mathrm{AI}(\mathit{patrow}) \rangle \\
\mathrm{AI}(\texttt{(}\ \mathit{pat}\ \texttt{)}) & = & \mathrm{AI}(\mathit{pat}) \\
 & & \\
\mathrm{AI} : \mathrm{PatRow} \to \mathrm{VarEnv} & & \\
\mathrm{AI}(\ldots) & = & \{\} \\
\mathrm{AI}(\mathit{lab}\ \texttt{=}\ \mathit{pat}\ \langle\ \texttt{,}\ \mathit{patrow} \rangle) & = & \mathrm{AI}(\mathit{pat})\ \langle + \mathrm{AI}(\mathit{patrow}) \rangle
\end{array}
$$

For declarations, AI has the form $\mathrm{Env} \times \mathrm{Dec} \to \mathrm{Bit} \times \mathrm{Env}$. The first component of the result indicates whether the declaration depends on specification constructs.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{Env} \times \mathrm{Dec} \to \mathrm{Bit} \times \mathrm{Env}} \\
\mathrm{AI}(E, \texttt{val}\ \mathit{valbind}) & = & (sp, VE \text{ in Env}) \\
 & & \text{where } (sp, VE) = \mathrm{AI}(E, \mathit{valbind}) \\
\mathrm{AI}(E, \texttt{datatype}\ \mathit{datbind}) & = & (\top, \mathrm{AI}(\mathit{datbind}) \text{ in Env}) \\
\mathrm{AI}(E, \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end}) & = & (sp, (VE \text{ in Env}) + E') \\
 & & \text{where } \left\{ \begin{array}{lcl} VE & = & \mathrm{AI}(\mathit{datbind}) \\ (sp, E') & = & \mathrm{AI}(E + VE, \mathit{dec}) \end{array} \right. \\
\mathrm{AI}(E, \texttt{exception}\ \mathit{exbind}) & = & (\top, \mathrm{AI}(\mathit{exbind}) \text{ in Env}) \\
\mathrm{AI}(E, \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end}) & = & (sp_1 \wedge sp_2, E_2) \\
 & & \text{where } \left\{ \begin{array}{lcl} (sp_1, E_1) & = & \mathrm{AI}(E, \mathit{dec}_1) \\ (sp_2, E_2) & = & \mathrm{AI}(E + E_1, \mathit{dec}_2) \end{array} \right. \\
 & & \\
\mathrm{AI}(E, \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n) & = & (\top, E_1 + \cdots + E_n) \\
 & & \text{where } E_i = E(\mathit{longstrid}_i),\ 1 \le i \le n \\
\mathrm{AI}(E, \mathit{dec}_1\ \langle ; \rangle\ \mathit{dec}_2) & = & (sp_1 \wedge sp_2, E_1 + E_2) \\
 & & \text{where } \left\{ \begin{array}{lcl} (sp_1, E_1) & = & \mathrm{AI}(E, \mathit{dec}_1) \\ (sp_2, E_2) & = & \mathrm{AI}(E + E_1, \mathit{dec}_2) \end{array} \right. \\
\mathrm{AI}(E, \quad) & = & (\top, \{\}) \\
 & & \\
\multicolumn{3}{l}{\mathrm{AI} : E \times \mathrm{ValBind} \to \mathrm{Bit} \times \mathrm{VarEnv}} \\
\mathrm{AI}(E, \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle) & = & (\mathrm{AI}(E, \mathit{exp})\ \langle \wedge\, sp \rangle,\ \mathrm{AI}(\mathit{pat})\ \langle + VE \rangle) \\
 & & \text{where } (sp, VE) = \mathrm{AI}(E, \mathit{valbind}) \\
\mathrm{AI}(E, \texttt{rec}\ \mathit{valbind}) & = & (sp, VE) \\
 & & \text{where } (sp, VE) = \mathrm{AI}(E + VE, \mathit{valbind})
\end{array}
$$

The cyclic dependency of VE in the definition of $\mathrm{AI}(E, \texttt{rec}\ \mathit{valbind})$ is unproblematic: we have $VE(id) = 1$ for all identifiers id defined in $\mathit{valbind}$.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{DatBind} \to \mathrm{VarEnv}} \\
\mathrm{AI}(\mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle) & = & \mathrm{AI}(\mathit{conbind})\ \langle + \mathrm{AI}(\mathit{datbind}) \rangle \\
 & & \\
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{ConBind} \to \mathrm{VarEnv}} \\
\mathrm{AI}(\mathit{con}\ \langle \texttt{|}\ \mathit{conbind} \rangle) & = & \{\mathit{id} \mapsto 1\}\ \langle + \mathrm{AI}(\mathit{conbind}) \rangle \\
 & & \text{where } \mathit{con} = \mathit{id}^{\mathtt{v}} \\
 & & \\
\multicolumn{3}{l}{\mathrm{AI} : \mathrm{ExBind} \to \mathrm{VarEnv}} \\
\mathrm{AI}(\mathit{excon}\ \langle \texttt{and}\ \mathit{exbind} \rangle) & = & \{\mathit{id} \mapsto 1\}\ \langle + \mathrm{AI}(\mathit{exbind}) \rangle \\
 & & \text{where } \mathit{excon} = \mathit{id}^{\mathtt{e}} \\
\mathrm{AI}(\mathit{excon}\ \texttt{=}\ \mathit{longexcon}\ \langle \texttt{and}\ \mathit{exbind} \rangle) & = & \{\mathit{id} \mapsto 1\}\ \langle + \mathrm{AI}(\mathit{exbind}) \rangle \\
 & & \text{where } \mathit{excon} = \mathit{id}^{\mathtt{e}}
\end{array}
$$

6.7 Inference Rules

The semantic rules allow sentences of the form

$$s, A \vdash \mathit{phrase} \Rightarrow A', s'$$

to be inferred, where A is usually an environment, A' is some semantic object and s,s' are the states before and after the evaluation represented by the sentence. Some hypotheses in rules are not of this form; they are called *side-conditions*.

In most rules the states s and s' are omitted from sentences; they are only included for those rules which are directly concerned with the state — either referring to its contents or changing it. When omitted, the convention for restoring them is as follows. If the rule is presented in the form

$$\frac{A_1 \vdash \mathit{phrase}_1 \Rightarrow A_1' \qquad A_2 \vdash \mathit{phrase}_2 \Rightarrow A_2' \quad \cdots \quad \cdots \quad A_n \vdash \mathit{phrase}_n \Rightarrow A_n'}{A \vdash \mathit{phrase} \Rightarrow A'}$$

then the full form is intended to be

$$\frac{s_0, A_1 \vdash \mathit{phrase}_1 \Rightarrow A_1', s_1 \qquad s_1, A_2 \vdash \mathit{phrase}_2 \Rightarrow A_2', s_2 \quad \cdots \quad \cdots \quad s_{n-1}, A_n \vdash \mathit{phrase}_n \Rightarrow A_n', s_n}{s_0, A \vdash \mathit{phrase} \Rightarrow A', s_n}$$

(Any side-conditions are left unaltered). Thus the left-to-right order of the hypotheses indicates the order of evaluation. Note that in the case $n = 0$, when there are no hypotheses (except possibly side-conditions), we have $s_n = s_0$; this implies that the rule causes no side effect. The convention is called the *state convention*, and must be applied to each version of a rule obtained by inclusion or omission of its options.

A second convention, the *exception convention*, is adopted to deal with the propagation of exception packets p. For each rule whose full form (ignoring side-conditions) is

$$\frac{s_1, A_1 \vdash \mathit{phrase}_1 \Rightarrow A_1', s_1' \qquad \cdots \qquad s_n, A_n \vdash \mathit{phrase}_n \Rightarrow A_n', s_n'}{s, A \vdash \mathit{phrase} \Rightarrow A', s'}$$

and for each k, $1 \leq k \leq n$, for which the result A_k' is not a packet p, an extra rule is added of the form

$$\frac{s_1, A_1 \vdash \mathit{phrase}_1 \Rightarrow A_1', s_1' \qquad \cdots \qquad s_k, A_k \vdash \mathit{phrase}_k \Rightarrow p', s'}{s, A \vdash \mathit{phrase} \Rightarrow p', s'}$$

where p' does not occur in the original rule.[6] This indicates that evaluation of phrases in the hypothesis terminates with the first whose result is a packet (other

[6] There is one exception to the exception convention in the definition of SML; no extra rule is added for rule 119 which deals with handlers, since a handler is the only means by which propagation of an exception can be arrested. For EML, rules 135 and 136 are also exempted from the exception convention — this is required so that a `NoCode` exception raised during the evaluation of *exp* in a ValBind of the form "*pat* `=` *exp* ⟨`and` *valbind*⟩" can be converted to the value `Incomplete`.

than one already treated in the rule), and this packet is the result of the phrase in the conclusion.

Recall from Section 1.2 that we support compound metavariables such as v/p. We also allow x/FAIL to range over $X \uplus \{\text{FAIL}\}$ where x ranges over X (and analogously for $x/\texttt{Incomplete}$); furthermore, we extend environment modification to allow for failure as follows:

$$VE + \text{FAIL} = \text{FAIL}.$$

Atomic Expressions

$\boxed{E \vdash atexp \Rightarrow v/p}$

$$\frac{}{E \vdash scon \Rightarrow \text{val}(scon)} \tag{103}$$

$$\frac{E(longvar) = v \qquad \text{AI}(v) = sp \qquad v \neq \texttt{Incomplete}}{s, E \vdash longvar \Rightarrow v, s \wedge sp} \tag{104}$$

$$\frac{E(longvar) = \texttt{Incomplete}}{s, E \vdash longvar \Rightarrow [\texttt{NoCode}], s_\perp} \tag{104.1}$$

$$\frac{E(longcon) = con}{E \vdash longcon \Rightarrow con} \tag{105}$$

$$\frac{E(longexcon) = en}{E \vdash longexcon \Rightarrow en} \tag{106}$$

$$\frac{\langle E \vdash exprow \Rightarrow r \rangle}{E \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + r \rangle \text{ in Val}} \tag{107}$$

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v} \tag{108}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow v} \tag{109}$$

$$\frac{}{s, E \vdash \texttt{?} \Rightarrow [\texttt{NoCode}], s_\perp} \tag{109.1}$$

Comments:

(104.1) When a variable's value is `Incomplete`, the variable evaluates to the packet [`NoCode`], indicating that no code exists for that binding.

(105) Value constructors denote themselves.

(106) Exception constructors are looked up in the exception environment component of E.

Expression Rows

$\boxed{E \vdash exprow \Rightarrow r/p}$

$$\frac{E \vdash exp \Rightarrow v \qquad \langle E \vdash exprow \Rightarrow r \rangle}{E \vdash lab \texttt{ = } exp \langle \texttt{ , } exprow \rangle \Rightarrow \{lab \mapsto v\} \langle + r \rangle} \tag{110}$$

Comment: We may think of components as being evaluated from left to right, because of the state and exception conventions.

Expressions

$\boxed{E \vdash exp \Rightarrow v/p}$

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \tag{111}$$

$$\frac{E \vdash exp \Rightarrow con \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (con, v)} \tag{112}$$

$$\frac{E \vdash exp \Rightarrow en \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (en, v)} \tag{113}$$

Deleted (`ref`application rule) (114)

Deleted (`:=` application rule) (115)

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \text{APPLY}(b, v) = v'/p}{E \vdash exp\ atexp \Rightarrow v'/p} \tag{116}$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \text{Rec}\, VE,\ v \vdash match \Rightarrow v' \end{array}}{E \vdash exp\ atexp \Rightarrow v'} \tag{117}$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \text{Rec}\, VE,\ v \vdash match \Rightarrow \text{FAIL} \end{array}}{E \vdash exp\ atexp \Rightarrow [\texttt{Match}]} \tag{118}$$

$$\frac{}{s, E \vdash exp_1^\bullet \texttt{ == } exp_2^\bullet \Rightarrow [\texttt{NoCode}], s_\perp} \tag{118.1}$$

$$\frac{}{s, E \vdash \texttt{exists } match^\bullet \Rightarrow [\texttt{NoCode}], s_\perp} \tag{118.2}$$

$$\frac{}{s, E \vdash \texttt{forall } match^\bullet \Rightarrow [\texttt{NoCode}], s_\perp} \tag{118.3}$$

$$\frac{}{s, E \vdash exp^{\bullet}\ \texttt{terminates} \Rightarrow [\texttt{NoCode}], s_{\perp}} \tag{118.4}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{119}$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad e \neq \texttt{NoCode} \qquad E, e \vdash match \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{120}$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad e \neq \texttt{NoCode} \qquad E, e \vdash match \Rightarrow \text{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [e]} \tag{121}$$

$$\frac{E \vdash exp \Rightarrow [\texttt{NoCode}]}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [\texttt{NoCode}]} \tag{121.1}$$

$$\frac{E \vdash exp \Rightarrow e}{E \vdash \texttt{raise}\ exp \Rightarrow [e]} \tag{122}$$

$$\frac{v = (match, E, \{\}) \qquad \text{AI}(v) = sp}{s, E \vdash \texttt{fn}\ match \Rightarrow v, s \wedge sp} \tag{123}$$

Comments:

(112)–(118) Note that none of the rules for function application has a premise in which the operator evaluates to a constructed value or a record. This is because we are interested in the evaluation of well-typed programs only, and in such programs *exp* will always have a functional type.

(116) The semantics of SML [MTH90] does not treat the case where an APPLY result is a packet. This is an oversight, probably caused by the fact that the exception convention does not apply to side-conditions.

Notice that the application of a basic value never raises the exception `NoCode` — in that case we would have to set the state flag to $\perp$ here.

(118.1)–(118.4) Remember that $s_{\perp}$ is shorthand for $(\perp, ens\ \text{of}\ s)$.

(119) The exception convention does not apply to this rule. If the operator evaluates to a packet then rule 120 or rule 121 or rule 121.1 must be used.

(121) Packets that are not handled by the *match* propagate.

(120),(121),(121.1) The packet [`NoCode`] cannot be handled.

(123) The third component of the closure is empty because the match does not introduce new recursively defined values.

Matches

$\boxed{E, v \vdash match \Rightarrow v'/p/\text{FAIL}}$

$$\frac{E, v \vdash mrule \Rightarrow v'}{E, v \vdash mrule\ \langle\ \texttt{|}\ match \rangle \Rightarrow v'} \tag{124}$$

$$\frac{E, v \vdash mrule \Rightarrow \text{FAIL}}{E, v \vdash mrule \Rightarrow \text{FAIL}} \tag{125}$$

$$\frac{E, v \vdash mrule \Rightarrow \text{FAIL} \qquad E, v \vdash match \Rightarrow v'/\text{FAIL}}{E, v \vdash mrule\ \texttt{|}\ match \Rightarrow v'/\text{FAIL}} \tag{126}$$

Comment: A value v occurs on the left of the turnstile, in evaluating a *match*. We may think of a *match* as being evaluated *against* a value; similarly, we may think of a pattern as being evaluated *against* a value. Alternative match rules are tried from left to right.

Match Rules

$\boxed{E, v \vdash mrule \Rightarrow v'/p/\text{FAIL}}$

$$\frac{E, v \vdash pat \Rightarrow VE \qquad E + VE \vdash exp \Rightarrow v'}{E, v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow v'} \tag{127}$$

$$\frac{E, v \vdash pat \Rightarrow \text{FAIL}}{E, v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow \text{FAIL}} \tag{128}$$

Declarations

$\boxed{E \vdash dec \Rightarrow E'/p}$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{val}\ valbind \Rightarrow VE \text{ in Env}} \tag{129}$$

$$\frac{\vdash datbind \Rightarrow VE}{E \vdash \texttt{datatype}\ datbind \Rightarrow VE \text{ in Env}} \tag{129.5}$$

$$\frac{\vdash datbind \Rightarrow VE \qquad E + VE \vdash dec \Rightarrow E'}{E \vdash \texttt{abstype}\ datbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow E'} \tag{129.6}$$

$$\frac{E \vdash exbind \Rightarrow VE}{E \vdash \texttt{exception}\ exbind \Rightarrow VE \text{ in Env}} \tag{130}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E_2} \tag{131}$$

$$\frac{E(longstrid_1) = E_1 \quad \cdots \quad E(longstrid_n) = E_n}{E \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n} \tag{132}$$

$$\frac{}{E \vdash \quad \Rightarrow \{\} \text{ in Env}} \tag{133}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash dec_1 \ \langle ; \rangle \ dec_2 \Rightarrow E_1 + E_2} \tag{134}$$

Value Bindings

$\boxed{E \vdash valbind \Rightarrow VE/p}$

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash pat \Rightarrow VE \qquad \langle E \vdash valbind \Rightarrow VE' \rangle}{E \vdash pat \texttt{ = } exp \ \langle \texttt{and } valbind \rangle \Rightarrow VE \ \langle + \ VE' \rangle} \tag{135}$$

$$\frac{E \vdash exp \Rightarrow \ [\texttt{NoCode}] \qquad E, \texttt{Incomplete} \vdash pat \Rightarrow VE \quad \langle E \vdash valbind \Rightarrow VE' \rangle}{E \vdash pat \texttt{ = } exp \ \langle \texttt{and } valbind \rangle \Rightarrow VE \ \langle + \ VE' \rangle} \tag{135.1}$$

$$\frac{E \vdash exp \Rightarrow \ [e], \ e \neq \texttt{NoCode}}{E \vdash pat \texttt{ = } exp \ \langle \texttt{and } valbind \rangle \Rightarrow [e]} \tag{135.2}$$

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat \texttt{ = } exp \ \langle \texttt{and } valbind \rangle \Rightarrow [\texttt{Bind}]} \tag{136}$$

$$\frac{E \vdash exp \Rightarrow \ [\texttt{NoCode}] \qquad E, \texttt{Incomplete} \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat \texttt{ = } exp \ \langle \texttt{and } valbind \rangle \Rightarrow [\texttt{NoCode}]} \tag{136.1}$$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{rec } valbind \Rightarrow \text{Rec}\, VE} \tag{137}$$

Comments:

(135),(136) The exception convention does not apply to these rules. If the expression evaluates to a packet then rule 135.1 or 135.2 (in the case of rule 135) or rule 136.1 (in the case of rule 136) must be used.

(135.1),(136.1) If the exception `NoCode` is raised while evaluating the expression (e.g. because it contains quantifiers), the exception is caught before doing the binding. Then the binding is done with the value `Incomplete`.

(135.2) Any exception other than `NoCode` is propagated as usual.

Data Type Bindings

$\boxed{\vdash datbind \Rightarrow VE}$

$$\frac{\vdash conbind \Rightarrow VE \qquad \langle \vdash datbind \Rightarrow VE' \rangle}{\vdash tyvarseq \ tycon \texttt{ = } conbind \ \langle \texttt{and } datbind \rangle \Rightarrow VE \ \langle + \ VE' \rangle} \tag{137.1}$$

Constructor Bindings

$\boxed{\vdash conbind \Rightarrow VE}$

$$\frac{con = id^{\mathsf{C}} \qquad \langle \vdash conbind \Rightarrow VE \rangle}{\vdash con\ \langle\ \mathsf{|}\ conbind \rangle \Rightarrow \{id \mapsto con\}\ \langle + VE \rangle} \tag{137.2}$$

Exception Bindings

$\boxed{E \vdash exbind \Rightarrow VE}$

In the SML definition, sentences for exception bindings have the more general form $E \vdash exbind \Rightarrow EE/p$. But the $/p$ is redundant, since packets can never arise here. VE now incorporates the information that was formerly in VE and EE because exception environments are not quite up to the task they were originally designed for, see [Kah93]. The distinction between exception constructors and other identifiers is handled in EML by the semantics of derived forms.

$$\frac{\begin{array}{c} en = \min(\mathrm{ExName} \setminus (ens\ \mathrm{of}\ s)) \qquad s' = s + \{en\} \\ excon = id^{\mathsf{E}} \qquad \langle s', E \vdash exbind \Rightarrow VE, s'' \rangle \end{array}}{s, E \vdash excon\ \langle \mathtt{and}\ exbind \rangle \Rightarrow \{id \mapsto en\} \langle + VE \rangle,\ s' \langle ' \rangle} \tag{138}$$

$$\frac{E(longexcon) = en \qquad excon = id^{\mathsf{E}} \qquad \langle E \vdash exbind \Rightarrow VE \rangle}{E \vdash excon\ \mathtt{=}\ longexcon\ \langle \mathtt{and}\ exbind \rangle \Rightarrow \{id \mapsto en\} \langle + VE \rangle} \tag{139}$$

Comments:

(138) The two side conditions ensure that a new exception name is generated and recorded as "used" in subsequent states. In contrast to Standard ML, the fresh exception name is chosen deterministically to conform with the verification semantics, see rules 256 and 257.

Atomic Patterns

$\boxed{E, v \vdash atpat \Rightarrow VE/\mathrm{FAIL}}$

$$\frac{}{E, v \vdash _ \Rightarrow \{\}} \tag{140}$$

$$\frac{v = \mathrm{val}(scon)}{E, v \vdash scon \Rightarrow \{\}} \tag{141}$$

$$\frac{v \neq \mathrm{val}(scon)}{E, v \vdash scon \Rightarrow \mathrm{FAIL}} \tag{142}$$

$$\frac{var = id^{\mathsf{V}}}{E, v \vdash var \Rightarrow \{id \mapsto v\}} \tag{143}$$

$$\frac{E(longcon) = v}{E, v \vdash longcon \Rightarrow \{\}} \tag{144}$$

$$\frac{E(longcon) \neq v}{E, v \vdash longcon \Rightarrow \text{FAIL}} \tag{145}$$

$$\frac{E(longexcon) = v}{E, v \vdash longexcon \Rightarrow \{\}} \tag{146}$$

$$\frac{E(longexcon) \neq v}{E, v \vdash longexcon \Rightarrow \text{FAIL}} \tag{147}$$

$$\frac{v = \{\}\langle +r\rangle \text{ in Val} \qquad \langle E, r \vdash patrow \Rightarrow VE/\text{FAIL}\rangle}{E, v \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow \{\}\langle +VE/\text{FAIL}\rangle} \tag{148}$$

$$\frac{v = \texttt{Incomplete} \qquad \langle E, \texttt{Incomplete} \vdash patrow \Rightarrow VE/\text{FAIL}\rangle}{E, v \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow \{\}\langle +VE/\text{FAIL}\rangle} \tag{148.1}$$

$$\frac{E, v \vdash pat \Rightarrow VE/\text{FAIL}}{E, v \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow VE/\text{FAIL}} \tag{149}$$

Comments:

(142),(145),(147) Any evaluation resulting in FAIL must do so because rule 142, rule 145, rule 147, rule 155, or rule 157 has been applied.

(148.1) The intention here (cf. rules 150.1, 151.1, 152.1) is that in a value binding of the form *pat* `=` `?` ⟨`and` *valbind*⟩ (or *pat* `=` *exp* ⟨`and` *valbind*⟩ where *exp* evaluates to [`NoCode`]), the undefined value decomposes arbitrarily as long as the type of *pat* ensures that any value of that type decomposes (disregarding the case of datatypes with only one constructor). So for example,

```
val (x,y) = ?
```

will successfully bind both `x` and `y` to `Incomplete` (recall that `(x,y)` expands to `{1=x,2=y}`), while

```
val [x] = ?
```

will fail (by the eventual use of rule 155) and consequently raise [`NoCode`].

Pattern Rows

$$\boxed{E, r/\texttt{Incomplete} \vdash \mathit{patrow} \Rightarrow \mathit{VE}/\text{FAIL}}$$

$$\frac{}{E, r \vdash \ldots \Rightarrow \{\}} \tag{150}$$

$$\frac{}{E, \texttt{Incomplete} \vdash \ldots \Rightarrow \{\}} \tag{150.1}$$

$$\frac{E, r(\mathit{lab}) \vdash \mathit{pat} \Rightarrow \text{FAIL}}{E, r \vdash \mathit{lab}\ \texttt{=}\ \mathit{pat}\ \langle\ \texttt{,}\ \mathit{patrow}\rangle \Rightarrow \text{FAIL}} \tag{151}$$

$$\frac{E, \texttt{Incomplete} \vdash \mathit{pat} \Rightarrow \text{FAIL}}{E, \texttt{Incomplete} \vdash \mathit{lab}\ \texttt{=}\ \mathit{pat}\ \langle\ \texttt{,}\ \mathit{patrow}\rangle \Rightarrow \text{FAIL}} \tag{151.1}$$

$$\frac{E, r(\mathit{lab}) \vdash \mathit{pat} \Rightarrow \mathit{VE} \qquad \langle E, r \vdash \mathit{patrow} \Rightarrow \mathit{VE}'/\text{FAIL}\rangle}{E, r \vdash \mathit{lab}\ \texttt{=}\ \mathit{pat}\ \langle\ \texttt{,}\ \mathit{patrow}\rangle \Rightarrow \mathit{VE}\langle+\ \mathit{VE}'/\text{FAIL}\rangle} \tag{152}$$

$$\frac{\begin{array}{c}E, \texttt{Incomplete} \vdash \mathit{pat} \Rightarrow \mathit{VE} \\ \langle E, \texttt{Incomplete} \vdash \mathit{patrow} \Rightarrow \mathit{VE}'/\text{FAIL}\rangle\end{array}}{E, \texttt{Incomplete} \vdash \mathit{lab}\ \texttt{=}\ \mathit{pat}\ \langle\ \texttt{,}\ \mathit{patrow}\rangle \Rightarrow \mathit{VE}\langle+\ \mathit{VE}'/\text{FAIL}\rangle} \tag{152.1}$$

Comments:

(151),(152) For well-typed programs *lab* will be in the domain of r.

Patterns

$$\boxed{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}}$$

$$\frac{E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}}{E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}} \tag{153}$$

$$\frac{\begin{array}{c}E(\mathit{longcon}) = \mathit{con} \qquad v = (\mathit{con}, v') \\ E, v' \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}\end{array}}{E, v \vdash \mathit{longcon}\ \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}} \tag{154}$$

$$\frac{E(\mathit{longcon}) = \mathit{con} \qquad v \notin \{\mathit{con}\} \times \text{Val}}{E, v \vdash \mathit{longcon}\ \mathit{atpat} \Rightarrow \text{FAIL}} \tag{155}$$

$$\frac{\begin{array}{c}E(\mathit{longexcon}) = \mathit{en} \qquad v = (\mathit{en}, v') \\ E, v' \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}\end{array}}{E, v \vdash \mathit{longexcon}\ \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}} \tag{156}$$

$$\frac{E(\mathit{longexcon}) = \mathit{en} \qquad v \notin \{\mathit{en}\} \times \text{Val}}{E, v \vdash \mathit{longexcon}\ \mathit{atpat} \Rightarrow \text{FAIL}} \tag{157}$$

Deleted (`ref` application rule) (158)

$$\frac{var = id^{\mathtt{V}} \qquad E, v \vdash pat \Rightarrow VE/\mathrm{FAIL}}{E, v \vdash var\ \mathtt{as}\ pat \Rightarrow \{id \mapsto v\} + VE/\mathrm{FAIL}} \tag{159}$$

7 Dynamic Semantics for Modules

The semantics of EML programs (see Section 10) does not depend on the dynamic semantics for Modules. The purpose of this section is (together with the dynamic semantics for the Core) to define computation with incomplete programs.

7.1 Reduced Syntax

Since signature expressions are mostly dealt with in the static semantics, the dynamic semantics need only take limited account of them. Unlike types, it cannot ignore them completely; the reason is that an explicit signature ascription plays the role of restricting the "view" of a structure — that is, restricting the domains of its component environments. However, the types and the sharing properties of structures and signatures are irrelevant to dynamic evaluation; the syntax is therefore reduced by the following transformations (in addition to those for the Core), for the purpose of the dynamic semantics of Modules:

- Qualifications "`of` *ty*" are omitted from constructor descriptions and exception descriptions.

- We remove specifications without computational content, i.e. any specification of the form "`axiom` *axdesc*", "`type` *typdesc*", "`eqtype` *typdesc*" or "`sharing` *shareq*" is replaced by the empty specification. Descriptions of datatypes cannot be replaced by the empty specification, as they also give rise to a variable environment in the static semantics. This was not correctly treated in the definition of SML [MTH90] or corrected in [MT91].

- The Modules phrase classes TypDesc, AxDesc, SpecExp and SharEq are omitted.

7.2 Semantic Objects

The semantic objects for the Modules dynamic semantics, extra to those for the Core dynamic semantics, are shown in Figure 16. An *interface* $I \in \mathrm{Int}$ represents a "view" of a structure. An interface in the dynamic semantics of SML includes a set of value variables and a set of exception constructors; they are replaced here by a status environment. Specifications and signature expressions will evaluate to interfaces; moreover, during the evaluation of a specification or signature expression, structures (to which a specification or signature expression may refer via "`open`") are represented only by their interfaces. To extract an interface from a dynamic environment we define two operations

$$\mathrm{Inter} \ : \ \begin{cases} \mathrm{Env} \to \mathrm{Int} \\ \mathrm{VarEnv} \to \mathrm{StatEnv} \end{cases}$$

$$
\begin{array}{rcl}
(strid : I, strexp : I', B) & \in & \text{FunctorClosure} \\
 & & \quad = (\text{StrId} \times \text{Int}) \times (\text{StrExp} \times \text{Int}) \times \text{Basis} \\
(IE, StE) \text{ or } I & \in & \text{Int} = \text{IntEnv} \times \text{StatEnv} \\
IE & \in & \text{IntEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Int} \\
st & \in & \text{Status} = \{\mathtt{v}, \mathtt{c}, \mathtt{e}\} \\
StE & \in & \text{StatEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Status} \\
G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Int} \\
F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunctorClosure} \\
(F, G, E) \text{ or } B & \in & \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
(G, IE) \text{ or } IB & \in & \text{IntBasis} = \text{SigEnv} \times \text{IntEnv}
\end{array}
$$

Figure 16: Further Semantic Objects

as follows:

$$\text{Inter}(SE, VE) \;=\; (IE, \text{Inter}\, VE)$$

where

$$
\begin{array}{rcl}
IE & = & \{strid \mapsto \text{Inter}\, E \;;\; SE(strid) = E\} \\
\text{Inter}\, VE & = & \{id \mapsto \mathtt{v} \;;\; id \in \text{Dom}\, VE\}
\end{array}
$$

Notice that *all* identifiers are assigned status `v` in Inter VE. This is so, because dynamic environments do not keep track of identifier status.

An *interface basis* $IB = (G, IE)$ is that part of a basis needed to evaluate signature expressions and specifications. The function Inter is extended to create an interface basis from a basis B as follows:

$$\text{Inter}(F, G, E) \;=\; (G, IE \text{ of } (\text{Inter}\, E))$$

A further operation

$$\downarrow : \text{Env} \times \text{Int} \to \text{Env}$$

is required, to cut down an environment E to a given interface I, representing the effect of an explicit signature ascription. It is defined as follows:

$$(SE, VE) \downarrow (IE, StE) \;=\; (SE', VE')$$

where

$$
\begin{array}{rcl}
SE' & = & \{strid \mapsto E \downarrow I \;;\; SE(strid) = E \text{ and } IE(strid) = I\} \\
VE' & = & \{id \mapsto v \;;\; VE(id) = v \text{ and } id \in \text{Dom}\, StE\}
\end{array}
$$

It is important to note that interfaces are statically known — they can be obtained by appropriate projections from the static value Σ of a principle signature and the environment E_{DER} produced by the semantics for derived forms, see Appendix B.

As in the dynamic semantics of the core language, the use of "?" in structure and functor bindings does not cause an exception to be raised until computation occurs which makes use of undefined components. Consider the following example:

```
structure A : SIG = ? ;
functor F(X : SIG) : SIG' = ? ;
structure B : SIG' = F(A)
```

This will successfully evaluate without raising an exception. Any later attempt to perform a computation using any component of `A` or `B` will raise an exception, however.

To generate a "trivial" dynamic environment from an interface (with respect to a given exception name set) we define the operation

$$\text{TrivEnv} : \text{Int} \times \text{ExNameSet} \to \text{Env} \times \text{ExNameSet}$$

as follows:

$$\text{TrivEnv}((IE, StE), ens) = (SE, VE), ens'$$

where

$$\begin{array}{ll} SE = & \{strid_1 \mapsto E_1, \ldots, strid_n \mapsto E_n\} \\ VE = & \{id \mapsto \texttt{Incomplete} \ ; \ StE(id) = \texttt{v}\}+ \\ & \{id \mapsto id^{\texttt{C}} \ ; \ StE(id) = \texttt{c}\}+ \\ & \{id_1 \mapsto en_1, \cdots, id_k \mapsto en_k \ ; \ StE(id_j) = \texttt{e}, 1 \leq j \leq k\} \\ ens' = & ens_n \cup \{en_1, \ldots, en_k\} \end{array}$$

where

$$\begin{array}{l} \{strid_1, \ldots, strid_n\} = \text{Dom}\, IE \\ VE(id) = en \wedge VE(id') = en \ \Rightarrow \ id = id' \\ \{en_1, \ldots, en_k\} \cap ens_n = \emptyset \\ (E_1, ens_1) = \text{TrivEnv}(IE(strid_1), ens) \\ (E_2, ens_2) = \text{TrivEnv}(IE(strid_2), ens_1) \\ \quad \cdots \qquad \cdots \\ (E_n, ens_n) = \text{TrivEnv}(IE(strid_n), ens_{n-1}) \end{array}$$

This binds `Incomplete` to each variable in the interface and a fresh exception name to each exception constructor. This operation is used to produce the dynamic environment needed in a structure binding of the form

$$strid \ : \ psigexp \ \texttt{=} \ \texttt{?} \ \langle \texttt{and} \ strbind \rangle .$$

Note that the result is independent (modulo the choice of exception names) of the particular enumeration of the domains of IE and StE.

To generate a "trivial" structure expression from an interface we define the operation

$$\text{TrivStrExp} : \text{Int} \to \text{StrExp}$$

as follows:

$$\begin{array}{l}\text{TrivStrExp}(IE, StE) = \ \texttt{struct} \\ \qquad \texttt{structure}\ strid_1\ \texttt{=}\ \text{TrivStrExp}(IE(strid_1)) \\ \qquad\qquad \cdots \quad \cdots \\ \qquad\qquad \texttt{and}\ strid_n\ \texttt{=}\ \text{TrivStrExp}(IE(strid_n)) \\ \qquad \texttt{val}\ var_1\ \texttt{=}\ \texttt{?}\ \texttt{and}\ \ldots\ \texttt{and}\ var_p\ \texttt{=}\ \texttt{?} \\ \qquad \texttt{exception}\ excon_1\ \texttt{and}\ \ldots\ \texttt{and}\ excon_m \\ \qquad \texttt{datatype dummy =}\ con_1\ \texttt{|}\ \ldots\ \texttt{|}\ con_r \\ \texttt{end}\end{array}$$

where:

$$\begin{array}{l}\{strid_1, \ldots, strid_n\} = \text{Dom}\, IE \\ \{var_1, \ldots, var_p\} = \{id^{\mathtt{v}} \mid id \in \text{Dom}\, StE\ ;\ StE(id) = \mathtt{v}\} \\ \{excon_1, \ldots, excon_m\} = \{id^{\mathtt{e}} \mid id \in \text{Dom}\, StE\ ;\ StE(id) = \mathtt{e}\} \\ \{con_1, \ldots, con_r\} = \{id^{\mathtt{c}} \mid id \in \text{Dom}\, StE\ ;\ StE(id) = \mathtt{c}\}\end{array}$$

This operation is used to produce the structure expression in the functor closure produced by a functor binding "*funid* (*strid* : *psigexp*) : *psigexp*′ = ?".

7.3 Inference Rules

The semantic rules allow sentences of the form

$$s, A \vdash phrase \Rightarrow A', s'$$

to be inferred, where A is either a basis or an interface basis or empty, A' is some semantic object and s,s' are the states before and after the evaluation represented by the sentence. Some hypotheses in rules are not of this form; they are called *side-conditions*.

The state and exception conventions are adopted as in the Core dynamic semantics. However, it may be shown that the only Modules phrases whose evaluation may cause a side-effect or generate an exception packet are of the form *strexp*, *strdec*, *strbind*, *sglstrbind* or *topdec*.

Structure Expressions $\boxed{B \vdash strexp \Rightarrow E/p}$

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash \texttt{struct}\ strdec\ \texttt{end} \Rightarrow E} \tag{160}$$

$$\frac{B(longstrid) = E}{B \vdash longstrid \Rightarrow E} \tag{161}$$

$$\frac{\begin{array}{c}B(funid) = (strid : I, strexp' : I', B') \\ B \vdash strexp \Rightarrow E \qquad B' + \{strid \mapsto E \downarrow I\} \vdash strexp' \Rightarrow E'\end{array}}{B \vdash funid\ (\ strexp\ ) \Rightarrow E' \downarrow I'} \tag{162}$$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B + E \vdash \mathit{strexp} \Rightarrow E'}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E'} \tag{163}$$

Comments:

(162) Before the evaluation of the functor body strexp', the actual argument E is cut down by the formal parameter interface I, so that any opening of *strid* resulting from the evaluation of strexp' will produce no more components than anticipated during the static elaboration.

Structure-level Declarations

$\boxed{B \vdash \mathit{strdec} \Rightarrow E/p}$

$$\frac{E \text{ of } B \vdash \mathit{dec} \Rightarrow E'}{B \vdash \mathit{dec} \Rightarrow E'} \tag{164}$$

$$\frac{}{s, B \vdash \texttt{axiom}\ \mathit{ax} \Rightarrow \{\} \text{ in Env}, s} \tag{164.1}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \mathit{SE} \text{ in Env}} \tag{165}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{166}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{167}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{168}$$

Structure Bindings

$\boxed{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}/p}$

$$\frac{B \vdash \mathit{sglstrbind} \Rightarrow \mathit{SE} \qquad \langle B \vdash \mathit{strbind} \Rightarrow \mathit{SE}' \rangle}{B \vdash \mathit{sglstrbind}\ \langle \texttt{and}\ \mathit{strbind} \rangle \Rightarrow \mathit{SE}\ \langle + \mathit{SE}' \rangle} \tag{169}$$

Single Structure Bindings

$\boxed{B \vdash \mathit{sglstrbind} \Rightarrow \mathit{SE}/p}$

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad \text{Inter}\, B \vdash \mathit{psigexp} \Rightarrow I}{B \vdash \mathit{strid} : \mathit{psigexp}\ \texttt{=}\ \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto E \downarrow I\}} \tag{169.1}$$

$$\frac{s, \text{Inter}\, B \vdash \mathit{psigexp} \Rightarrow I, s' \qquad (E, \mathit{ens}') = \text{TrivEnv}(I, \mathit{ens} \text{ of } s')}{s, B \vdash \mathit{strid} : \mathit{psigexp}\ \texttt{=}\ \texttt{?} \Rightarrow \{\mathit{strid} \mapsto E\}, (\mathit{sp} \text{ of } s', \mathit{ens}')} \tag{169.2}$$

$$\frac{B \vdash strexp \Rightarrow E}{B \vdash strid \texttt{ = } strexp \Rightarrow \{strid \mapsto E\}} \tag{169.3}$$

Comments:

(169.1),(169.2) As in the static semantics, *psigexp* constrains the "view" of the structure. The restriction must be done in the dynamic semantics to ensure that any dynamic opening of the structure produces no more components than anticipated during the static elaboration.

(169.2) The state has been made explicit here because the side condition accesses the state.

Signature Expressions

$\boxed{IB \vdash sigexp \Rightarrow I}$

$$\frac{IB \vdash spec \Rightarrow I}{IB \vdash \texttt{sig } spec \texttt{ end} \Rightarrow I} \tag{170}$$

$$\frac{IB(sigid) = I}{IB \vdash sigid \Rightarrow I} \tag{171}$$

Principal Signatures

$\boxed{IB \vdash psigexp \Rightarrow I}$

$$\frac{IB \vdash sigexp \Rightarrow I}{IB \vdash sigexp \Rightarrow I} \tag{171.1}$$

Signature Declarations

$\boxed{IB \vdash sigdec \Rightarrow G}$

$$\frac{IB \vdash sigbind \Rightarrow G}{IB \vdash \texttt{signature } sigbind \Rightarrow G} \tag{172}$$

$$\frac{}{IB \vdash \quad \Rightarrow \{\}} \tag{173}$$

$$\frac{IB \vdash sigdec_1 \Rightarrow G_1 \qquad IB + G_1 \vdash sigdec_2 \Rightarrow G_2}{IB \vdash sigdec_1 \ \langle ; \rangle \ sigdec_2 \Rightarrow G_1 + G_2} \tag{174}$$

Signature Bindings

$\boxed{IB \vdash sigbind \Rightarrow G}$

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash sigbind \Rightarrow G \rangle}{IB \vdash sigid \texttt{ = } psigexp \ \langle \texttt{and } sigbind \rangle \Rightarrow \{sigid \mapsto I\} \ \langle + G \rangle} \tag{175}$$

Specifications

$\boxed{IB \vdash spec \Rightarrow I}$

$$\frac{\vdash valdesc \Rightarrow StE}{IB \vdash \texttt{val}\ valdesc \Rightarrow StE \text{ in Int}} \tag{176}$$

$$\frac{\vdash datdesc \Rightarrow StE}{IB \vdash \texttt{datatype}\ datdesc \Rightarrow StE \text{ in Int}} \tag{176.1}$$

$$\frac{\vdash exdesc \Rightarrow StE}{IB \vdash \texttt{exception}\ exdesc \Rightarrow StE \text{ in Int}} \tag{177}$$

$$\frac{IB \vdash strdesc \Rightarrow IE}{IB \vdash \texttt{structure}\ strdesc \Rightarrow IE \text{ in Int}} \tag{178}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow I_2} \tag{179}$$

$$\frac{IB(longstrid_1) = I_1 \quad \cdots \quad IB(longstrid_n) = I_n}{IB \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow I_1 + \cdots + I_n} \tag{180}$$

$$\frac{IB(sigid_1) = I_1 \quad \cdots \quad IB(sigid_n) = I_n}{IB \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow I_1 + \cdots + I_n} \tag{181}$$

$$\frac{}{IB \vdash \qquad \Rightarrow \{\} \text{ in Int}} \tag{182}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash spec_1\ \langle ; \rangle\ spec_2 \Rightarrow I_1 + I_2} \tag{183}$$

Comments:

(176.1) In the definition of SML, datatype descriptions are treated as empty specifications. This is a bug since they elaborate to non-empty variable environments in the static semantics.

(179),(183) Note that StE of I_1 is not needed for the evaluation of $spec_2$.

Value Descriptions

$\boxed{\vdash valdesc \Rightarrow StE}$

$$\frac{var = id^{\mathtt{v}} \qquad \langle \vdash valdesc \Rightarrow StE \rangle}{\vdash var\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{id \mapsto \mathtt{v}\}\ \langle +StE \rangle} \tag{184}$$

Datatype Descriptions

$\boxed{\vdash datdesc \Rightarrow StE}$

$$\frac{\vdash condesc \Rightarrow StE \qquad \langle \vdash datdesc \Rightarrow StE' \rangle}{\vdash tyvarseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow StE\ \langle +StE' \rangle} \tag{184.1}$$

Constructor Descriptions $\boxed{\vdash condesc \Rightarrow StE}$

$$\frac{con = id^{\mathtt{c}} \qquad \langle \vdash condesc \Rightarrow StE \rangle}{\vdash con\ \langle\ |\ condesc \rangle \Rightarrow \{id \mapsto \mathtt{c}\}\ \langle + StE \rangle} \tag{184.2}$$

Exception Descriptions $\boxed{\vdash exdesc \Rightarrow StE}$

$$\frac{excon = id^{\mathtt{e}} \qquad \langle \vdash exdesc \Rightarrow StE \rangle}{\vdash excon\ \langle \mathtt{and}\ exdesc \rangle \Rightarrow \{id \mapsto \mathtt{e}\}\ \langle + StE \rangle} \tag{185}$$

Structure Descriptions $\boxed{IB \vdash strdesc \Rightarrow IE}$

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash strdesc \Rightarrow IE \rangle}{IB \vdash strid\ :\ sigexp\ \langle \mathtt{and}\ strdesc \rangle \Rightarrow \{strid \mapsto I\}\ \langle + IE \rangle} \tag{186}$$

Functor Bindings $\boxed{B \vdash funbind \Rightarrow F}$

$$\frac{\begin{array}{c}\mathrm{Inter}\, B \vdash psigexp \Rightarrow I \qquad \mathrm{Inter}\, B + \{strid \mapsto I\} \vdash psigexp' \Rightarrow I' \\ \langle B \vdash funbind \Rightarrow F \rangle\end{array}}{\begin{array}{l}B \vdash funid\ \mathtt{(}\ strid\ :\ psigexp\ \mathtt{)}\ :\ psigexp'\ \mathtt{=}\ strexp\ \langle \mathtt{and}\ funbind \rangle \Rightarrow \\ \qquad \{funid \mapsto (strid : I, strexp : I', B)\}\ \langle + F \rangle\end{array}} \tag{187}$$

$$\frac{\begin{array}{c}\mathrm{Inter}\, B \vdash psigexp \Rightarrow I \qquad \mathrm{Inter}\, B + \{strid \mapsto I\} \vdash psigexp' \Rightarrow I' \\ \langle B \vdash funbind \Rightarrow F \rangle\end{array}}{\begin{array}{l}B \vdash funid\ \mathtt{(}\ strid\ :\ psigexp\ \mathtt{)}\ :\ psigexp'\ \mathtt{=\ ?}\ \langle \mathtt{and}\ funbind \rangle \Rightarrow \\ \qquad \{funid \mapsto (strid : I, \mathrm{TrivStrExp}\, I' : I', B)\}\ \langle + F \rangle\end{array}} \tag{187.1}$$

Functor Declarations $\boxed{B \vdash fundec \Rightarrow F}$

$$\frac{B \vdash funbind \Rightarrow F}{B \vdash \mathtt{functor}\ funbind \Rightarrow F} \tag{188}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}} \tag{189}$$

$$\frac{B \vdash fundec_1 \Rightarrow F_1 \qquad B + F_1 \vdash fundec_2 \Rightarrow F_2}{B \vdash fundec_1\ \langle ; \rangle\ fundec_2 \Rightarrow F_1 + F_2} \tag{190}$$

Top-level Declarations

$\boxed{B \vdash topdec \Rightarrow B'/p}$

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash strdec \Rightarrow E \text{ in Basis}} \quad (191)$$

$$\frac{\text{Inter}\, B \vdash sigdec \Rightarrow G}{B \vdash sigdec \Rightarrow G \text{ in Basis}} \quad (192)$$

$$\frac{B \vdash fundec \Rightarrow F}{B \vdash fundec \Rightarrow F \text{ in Basis}} \quad (193)$$

8 Verification Semantics for the Core

While the dynamic semantics defines evaluation of an expression in an *environment*, the verification semantics defines its evaluation in a *model*, which consists of an environment, a trace, and an interpretation for question marks, see below. We do not require *a priori* that the question mark interpretation is well-formed in the sense that the replacement of question marks leads to a well-typed program; such ill-formed models are eliminated *a posteriori* during verification.

Another difference to the dynamic semantics is that some rules in the verification semantics are non-computational, involving e.g. infinitary premises (expressed using higher-order rules), which is necessary to give meaning to specification constructs.

8.1 Semantic Objects

Many semantic objects used in the verification semantics already occur in the static and dynamic semantics. We use the same names for these object classes and the same variables to range over them, except when ambiguities occur. In that case, we attach to variables (similarly for object classes) a subscript *STAT* (resp. *DYN*), to indicate that it ranges over the corresponding object class of the static (resp. dynamic) semantics.

Additional and modified semantic objects are shown in Figure 17.

The conventions and notations used in earlier sections are adopted here as well; for example projection, injection and modification retain the meaning they were given in Section 4.3.

For *type values*, we have instantiation, equivalence, and closure analogous to type schemes and trace schemes: $\forall\alpha^{(k)}.(v,\tau) \succ (v',\tau')$ if there is a type substitution ϑ with $\mathrm{Dom}\,\vartheta = \alpha^{(k)}$ and $\vartheta(\tau) = \tau'$, $\vartheta(v) = v'$. (In general, $\vartheta(v) = v$ does not hold, because values can contain closures and closures can contain type information in traces and environments.) $\mathrm{Clos}_C(v,\tau) = \forall\alpha^{(k)}.(v,\tau)$, where $\alpha^{(k)}$ are the variables occurring in v or τ but not in C. Closure of variable environments is also analogous to the closure of static variable environments, e.g. $\mathrm{Clos}_{FE}VE$ can be obtained from VE by pointwise abstracting all type variables not free in FE.

We use the notation $M(x)$ or $FE(x)$ to apply the appropriate component of E of FE to x, where x is a (possibly long) identifier of some sort. The application of environments to long identifiers is analogous to its definition in Section 4.3.

8.2 Question Mark Interpretation

A *question mark interpretation* QI consists of two maps, QIE (for expressions) and QIT (for types). These maps are not much more than (infinite) lists of pieces of syntax (of the bare language), intended as replacements for occurrences of question marks in expressions and type bindings. The well-formedness of these

$$
\begin{array}{rcl}
n & \in & \mathcal{N} = \{0, 1, 2, ...\} \\
CT & \in & \text{TyCons} = \text{Fin}(\text{Id} \times \text{TypeScheme}) \\
ET & \in & \text{TyExcs} = \text{Fin}(\text{ExName} \times \text{Type}) \\
VT & \in & \text{ValTemp} = \text{TyCons} \times \text{TyExcs} \\
TI \text{ or } \varphi_{\text{Ty}} & \in & \text{TypeInt} = \text{TyName} \to \text{TypeFcn} \\
QI & \in & \text{QInt} = \text{QIntExp} \times \text{QIntTy} \\
QIE & \in & \text{QIntExp} = \mathcal{N} \to \text{Exp} \\
QIT & \in & \text{QIntTy} = \mathcal{N} \to \text{Ty} \\
(ens, TI, \varphi_{\text{Ty}}, VT, n) \text{ or } s & \in & \text{State} = \text{ExNameSet} \times \text{TypeInt} \times \text{TypeInt} \\
 & & \quad \times \text{ValTemp} \times \mathcal{N} \\
(SE, TE, VE) \text{ or } E & \in & \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{VarEnv} \\
SE & \in & \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Str} \\
VE & \in & \text{VarEnv} = \text{Id} \xrightarrow{\text{fin}} \text{TypeVal} \\
\forall \alpha^{(k)}.(v, \tau) \text{ or } tv & \in & \text{TypeVal} = \uplus_{k \geq 0} \text{TyVar}^k \times \text{Val} \times \text{Type} \\
(E, m) \text{ or } S & \in & \text{Str} = \text{Env} \times \text{StrName} \\
v & \in & \text{Val} = \text{SVal} \uplus \text{BasVal} \uplus \text{Con} \uplus (\text{Con} \times \text{Val}) \\
 & & \quad \uplus \text{ExVal} \uplus \text{Record} \uplus \text{Closure} \\
r & \in & \text{Record} = \text{Lab} \xrightarrow{\text{fin}} \text{Val} \\
e & \in & \text{ExVal} = \text{ExName} \uplus (\text{ExName} \times \text{Val}) \\
[e] \text{ or } p & \in & \text{Pack} = \text{ExVal} \\
(match, M, VE) & \in & \text{Closure} = \text{Match} \times \text{Mod} \times \text{VarEnv} \\
(E, QI) \text{ or } FE & \in & \text{FullEnv} = \text{Env} \times \text{QInt} \\
(FE, \gamma) \text{ or } M & \in & \text{Mod} = \text{FullEnv} \times \text{Trace}
\end{array}
$$

Figure 17: Semantic Objects

replacements w.r.t. the context in which they occur is checked during verification, for instance rule 201 checks that the replacement of a question mark in an expression has the right type.

We use the state (see next section) to associate the elements of these lists with occurrences of question marks.

8.3 State

A state s has five components: a set of exception names *ens*, for exactly the same purpose as in the dynamic semantics; a type interpretation TI for modelling the interpretation of type names originating from question mark type bindings; another type interpretation φ_{Ty} for representing the matching realisations of abstract types in signatures; a pair of sets of value templates VT for maintaining some necessary additional information about constructors — see the next section; and, a natural number n for counting occurrences of `?`. We write $s \oplus n$ for the state in which n of s is *replaced* by n.

The component n of s is always statically known; in other words, using the state is not essential for finding the right interpretations for question mark occurrences. An alternative would be to change the syntax of the bare language, requiring a (unique) label for each occurrence of `?`; the semantics of derived forms could be used to compute such labels. However, the method we use here seems to make less "noise".

We define a family of functions Replace : $(\mathrm{Phrase} \times \mathrm{QInt} \times \mathcal{N}) \to (\mathrm{Phrase} \times \mathcal{N})$, where Phrase is a class of syntactical phrases (of the core). These functions replace all occurrences of `?` by the corresponding object taken from the question mark interpretation. The purpose of the natural number is to establish this correspondence. For example, we have for atomic expressions:

$$\begin{aligned}
\mathrm{Replace}(\texttt{?}, QI, n) &= ((QIE \text{ of } QI)(n), n+1) \\
\mathrm{Replace}(\mathit{longvar}, QI, n) &= (\mathit{longvar}, n) \\
\mathrm{Replace}(\texttt{let } \mathit{dec} \texttt{ in } \mathit{exp} \texttt{ end}, QI, n) &= (\texttt{let } \mathit{dec}' \texttt{ in } \mathit{exp}' \texttt{ end}, n'') \text{ where} \\
&\qquad (\mathit{dec}', n') = \mathrm{Replace}(\mathit{dec}, QI, n) \\
&\qquad (\mathit{exp}', n'') = \mathrm{Replace}(\mathit{exp}, QI, n')
\end{aligned}$$

On other syntactical phrases, Replace is defined completely analogously, following the schema of `let`-expressions for all composed phrases. $\mathrm{Replace}(\mathit{phrase}, QI, n)$ can be obtained from *phrase* by replacing the $k+1$-th occurrence of `?` (an atomic expression or in a type binding) by either $(QIE \text{ of } QI)(n+k)$ or $(QIT \text{ of } QI)(n+k)$, respectively.

8.4 Value Access

The VT component of the state contains the *value templates*; these are all value constructors and exception names introduced so far. Any value of any type can be

decomposed into value templates, basic values, special values, and closures. The purpose of storing the value templates is to allow quantifiers to access values that are "hidden" due to e.g. value constructors used to build them being no longer in scope.

We define a semantic function Comp : Env × ValTemp → VarEnv for completing environments as follows: $\mathrm{Comp}(E, VT) = VE$ where

1. For each (en, τ) in VT there is an id such that $VE(id) = (en, \tau)$.
2. For each $(id, \forall\alpha^{(k)}.\tau)$ in VT there is an id' such that $VE(id') = \forall\alpha^{(k)}.(id^{\mathsf{C}}, \tau)$.
3. Any id in the domain of VE is of one of the two above forms and is not in the domain of E.

We abbreviate $\mathrm{Comp}(E \text{ of } FE, VT \text{ of } s)$ as $\mathrm{Comp}(FE, s)$.

The intention is to enable each value "belonging to a type" to be obtained by verificating some expression in $\mathrm{Comp}(FE, s)$.

To convert constructor environments to value templates, we assume another semantic function Graph, which maps a finite function to its graph (set of pairs). In particular, Graph maps a constructor environment CE to a finite set of value templates CT.

8.5 Type Interpretation

Type interpretations are the same as type realisations (see Section 5.6), but they have a wider application in EML than type realisations have in the semantics of SML, therefore the different name. A type realisation TI provides a way of translating (semantic) types, for example to interpret ?-types or to translate between a concrete and an abstract type. Given a type interpretation TI, we define a function $TI^{\#}$ between types as follows:

$$TI^{\#}(\tau) = \tau\{TI(t^{(k)})/t^{(k)} \ ; \ t^{(k)} \in \mathrm{Dom}\, TI\}$$

For the notation, see Section 4.4. We shall also apply $TI^{\#}$ to other semantic objects, meaning the simultaneous application of $TI^{\#}$ to all components.

A state s contains two type interpretation, TI and φ_{Ty}. The purpose of TI is to interpret question mark types, i.e. to replace type names that were chosen during static analysis for unknown types by their replacement in the given model. φ_{Ty} is the type realisation for abstract types in signatures. We abbreviate $(TI \text{ of } s)^{\#}$ as $s^{\#}$ and $(\varphi_{\mathrm{Ty}} \text{ of } s)^{\#} \circ s^{\#}$ as $s^{\#\#}$.

Static information stored in traces is typically model-independent, having been obtained from the static analysis of phrases. When interpreting this information in a given model we have to replace type names generated by question mark type bindings (rule 28.1) by the concrete types the model provides. This explains the frequent use of $s^{\#}$ in the rules. With $s^{\#\#}$ we can access the underlying structure of a type, i.e. how it is "implemented"; we use this for quantification and comparison of values.

8.6 Projections to Dynamic and Static Semantics

The notion of value is slightly different from that in the dynamic semantics because models are part of closures. We define a family of functions Dyn mapping semantic objects of the verification semantics to the corresponding semantic objects of the dynamic semantics having the same name, as follows:

$$
\begin{array}{rcl}
\pi_i(x_1, \cdots, x_i, \cdots, x_n) & = & x_i \\
\mathrm{Dyn}(SE, TE, VE) & = & (\mathrm{Dyn}\, SE, \mathrm{Dyn}\, VE) \\
\mathrm{Dyn}\, VE & = & \mathrm{Dyn} \circ VE \\
\mathrm{Dyn}\, SE & = & \mathrm{Dyn} \circ \pi_1 \circ SE \\
\mathrm{Dyn}(\forall \alpha^{(k)}.(v, \tau)) & = & \mathrm{Dyn}\, v \\
\mathrm{Dyn}\, v & = & v, \quad v \in \mathrm{SVal} \uplus \mathrm{BasVal} \uplus \mathrm{Con} \uplus \mathrm{ExName} \\
\mathrm{Dyn}(v, v') & = & (v, \mathrm{Dyn}\, v'), \quad v \in \mathrm{Con} \uplus \mathrm{ExName} \\
\mathrm{Dyn}\, r & = & \mathrm{Dyn} \circ r \\
\mathrm{Dyn}[e] & = & [\mathrm{Dyn}\, e] \\
\mathrm{Dyn}(match, M, VE) & = & (\mathrm{Replace}(match, QI \text{ of } M, 0), \mathrm{Dyn}\, M, \mathrm{Dyn}\, VE) \\
\mathrm{Dyn}\, M & = & \mathrm{Dyn}(FE \text{ of } M) \\
\mathrm{Dyn}\, FE & = & \mathrm{Dyn}(E \text{ of } FE) \\
\mathrm{Dyn}\, s & = & (\top, ens \text{ of } s)
\end{array}
$$

The function Replace is used to replace question marks in closure values. It is assumed here that the question mark interpretation in a closure always indexes its occurrences of ? from 0. The function Rec is extended to variable environments of the verification semantics in the obvious way, i.e. it unrolls closure values analogously as in the dynamic semantics (without changing any types).

We define another family of functions Stat, mapping objects of the verification semantics to corresponding objects in the static semantics. It is defined as follows:

$$
\begin{array}{rcl}
\mathrm{Stat}(SE, TE, VE) & = & (\mathrm{Stat}\, SE, TE, \mathrm{Stat}\, VE) \\
\mathrm{Stat}\, VE & = & \mathrm{Stat} \circ VE \\
\mathrm{Stat}(\forall \alpha^{(k)}.(v, \tau)) & = & \forall \alpha^{(k)}.\tau \\
\mathrm{Stat}\, SE & = & \{strid \mapsto (m, \mathrm{Stat}\, E) \ ; \ SE(strid) = (E, m)\}
\end{array}
$$

8.7 Relationship to Dynamic Semantics

If any expression $exp^\bullet$ evaluates in the dynamic semantics to a value or packet, i.e. $s_{\mathrm{DYN}}, E_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} exp^\bullet \Rightarrow v_{\mathrm{DYN}}/p_{\mathrm{DYN}}, s'_{\mathrm{DYN}}$ without changing the flag, i.e. sp of $s_{\mathrm{DYN}} = sp$ of $s'_{\mathrm{DYN}} = \top$, then for any s such that $s_{\mathrm{DYN}} = \mathrm{Dyn}\, s$ and for any M such that $E_{\mathrm{DYN}} = \mathrm{Dyn}\, M$ we also have $s, M \vdash exp^\bullet \Rightarrow v/p, s'$ such that $\mathrm{Dyn}(v/p) = v_{\mathrm{DYN}}/p_{\mathrm{DYN}}$ and $\mathrm{Dyn}\, s' = s'_{\mathrm{DYN}}$, provided the trace γ of M is well-formed for exp.

Dynamic and verification semantics differ on expressions only when the dynamic semantics sets the *sp* component to $\perp$. Even in that case, the differences are not substantial. In particular, any non-terminating evaluation in the dynamic semantics corresponds to non-termination in the verification semantics.

This close relationship between verification semantics and dynamic semantics is due to the fact that the rules differ significantly only in places where either the *sp* component of the state in the dynamic semantics is set to $\perp$, or where packets (in particular [`NoCode`]) are treated that do not occur in the corresponding place in the verification semantics. The packet [`NoCode`] cannot be handled: `NoCode` is a basic exception name not associated with any identifier, see Section 6.5, and furthermore it cannot be matched by a variable pattern in a handler, see rules 120–121.1 of the dynamic semantics. The dynamic semantics provides only one place to capture [`NoCode`]: rule 135.1, for value declarations. This means that the dynamic semantics can ignore `NoCode`, provided it ignores the declaration that raises it, since using an identifier bound to `Incomplete` raises `NoCode` again, rule 104.1.

The evaluation of an expression containing a question mark, when it yields a value, can differ from its corresponding verification; this is the case when the question mark interpretation maps a `?` occurrence to an expression which raises an exception. Whenever the verification semantics makes reference to the dynamic semantics, it is with a phrase that does not contain question marks.

One possible result when verificating $exp^\bullet$ `terminates` is the packet [`Abuse`], see rule 216. In this case the evaluation of $exp^\bullet$ terminated successfully, but *sp* was set to $\perp$. Thus successful evaluation does not guarantee successful verification — the packet [`Abuse`] indicates that we have not obtained reliable information about the termination of the verification of $exp^\bullet$.

8.8 Sentences of Static and Dynamic Semantics

Another difference between dynamic and verification semantics is that the latter makes explicit use of the results of the static semantics; in particular it typechecks certain expressions using contexts produced in the course of static analysis.

These contexts are explicitly provided by the γ component of a model. The reader may observe that traces are decomposed in the verification semantics in the same way as they are composed in the static semantics.

To distinguish sentences of the static and dynamic semantics from those of the verification semantics, we supply the $\vdash$ with an appropriate subscript in the former case, that is $\vdash_{\mathrm{STAT}}$ or $\vdash_{\mathrm{DYN}}$, respectively.

8.9 Inference Rules

The semantic rules allow sentences of the form $s, A \vdash \mathit{phrase} \Rightarrow A', s'$ to be inferred, where A is usually a model, A' is some semantic object and s, s' are the states before and after verification represented by the sentence. Additionally, we

have two other forms of sentences, equality sentences $s, FE, \gamma \vdash exp_1^\bullet \equiv exp_2^\bullet \Rightarrow v$, and comparison sentences $s, FE, C, \gamma \vdash v \approx v' \Rightarrow v''$. Beside these sentences, we allow other hypotheses as in the dynamic semantics, including hypotheses in the form of sentences of the static or dynamic semantics as explained in Section 8.8. *All* these other hypotheses have the status of side-conditions; this includes higher-order rules.

In most rules the states s and s' are omitted from sentences. The state and exception convention are adopted as in the dynamic semantics for the Core, except that "state" refers there to a different semantic object.

The exception convention does not apply to rules 210–218 (those for the equality predicate `==`, quantifiers, `terminates` and the exception handler) nor to rules 224–232 (equality and comparison rules).

Atomic Expressions

$\boxed{M \vdash atexp \Rightarrow v/p}$

$$\frac{}{M \vdash scon \Rightarrow \mathrm{val}(scon)} \tag{194}$$

$$\frac{FE(longvar) \succ (v, s^\#(\tau)) \qquad \dfrac{FE(longvar) \succ (v', s^\#(\tau))}{\exists \vartheta.\, \vartheta(v) = v' \wedge \mathrm{Dom}\,\vartheta \cap \mathrm{tyvars}\, FE = \emptyset}}{s, (FE, \tau) \vdash longvar \Rightarrow v, s} \tag{195}$$

$$\frac{M(longcon) \succ (con, \tau)}{M \vdash longcon \Rightarrow con} \tag{196}$$

$$\frac{M(longexcon) = (en, \tau)}{M \vdash longexcon \Rightarrow en} \tag{197}$$

$$\frac{\langle (FE, \gamma) \vdash exprow \Rightarrow r \rangle}{(FE, \epsilon\langle\cdot\gamma\rangle) \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + r\rangle \text{ in Val}} \tag{198}$$

$$\frac{(FE, \gamma) \vdash dec \Rightarrow E' \qquad (FE + E', \gamma') \vdash exp \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v} \tag{199}$$

$$\frac{M \vdash exp \Rightarrow v}{M \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow v} \tag{200}$$

$$\frac{\begin{array}{c} n = n \text{ of } s \qquad (QIE \text{ of } FE)(n) = exp^\bullet \\ s^\#(C) \vdash_{\mathrm{STAT}} exp^\bullet \Rightarrow \tau', \emptyset, \gamma \qquad s^\#(\tau) = \tau' \\ s \oplus (n+1), (FE, \gamma) \vdash exp^\bullet \Rightarrow v, s' \end{array}}{s, (FE, (C, \tau)) \vdash \texttt{?} \Rightarrow v, s'} \tag{201}$$

Comments:

(195) Types in verification environments have already been interpreted with an appropriate type interpretation TI, but τ, which comes from the elaboration, has not been so interpreted.

Applying a type substitution to a value has an effect only if the value contains closures and the closures contain free type variables in the domain of the type substitution. The type substitution is typically determined by $s^{\#}(\tau)$, but one can construct pathological examples for which this is not true. The side-condition excludes such a choice to keep verification deterministic.

The value `Incomplete` does not appear in the verification semantics; therefore we have no rule corresponding to 104.1 (same for rules 135.1, 136.1, 148.1, 150.1, 151.1 and 152.1).

(201) The model substitutes each occurrence of `?` by a fixed expression, preferably of the appropriate type. The type is *appropriate* if it is equal to the type of `?` *after* taking the interpretation of types into consideration. We do not get appropriateness of types for free, thus the side-condition $s^{\#}(\tau) = \tau'$ which excludes unwanted models.

By using an *expression* rather than a *value* we implicitly use a kind of second-order substitution, because the value of this expression depends on the context. In fact each `?` in an expression can roughly be seen as an abbreviation for $?(x_1, ..., x_n)$ where the x_i are the variables bound by the context in which `?` occurs. Here, `?` could indeed be replaced by a *value*, a n-ary function.

Notice that there is a subtle problem with the type of $exp^{\bullet}$. We require that $exp^{\bullet}$ has exactly the same type as the particular occurrence of `?`; otherwise the rule is not applicable. In practice, this means that `?` almost always has to be qualified by its intended type. The reason for requiring equality of types rather than allowing instances is type safety, i.e. we may have used an instance (or even different instances) of a polymorphic object that was defined using `?`. Since such instances only *may* be in conflict with the type of $exp^{\bullet}$, this is not necessarily harmful, but the alternative to the rule given would be to make the static context dependent on the model's choice for question marks, and hence to have a separate static semantics for each model.

Expression Rows

$\boxed{M \vdash exprow \Rightarrow r/p}$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow v \qquad \langle (FE, \gamma') \vdash exprow \Rightarrow r \rangle}{(FE, \gamma\langle \cdot \gamma' \rangle) \vdash lab \texttt{ = } exp \langle \texttt{ , } exprow \rangle \Rightarrow \{lab \mapsto v\}\langle + r \rangle} \tag{202}$$

Expressions

$$\boxed{M \vdash exp \Rightarrow v/p}$$

$$\frac{M \vdash atexp \Rightarrow v}{M \vdash atexp \Rightarrow v} \tag{203}$$

$$\frac{(FE,\gamma) \vdash exp \Rightarrow con \qquad (FE,\gamma') \vdash atexp \Rightarrow v}{(FE,\gamma \cdot \gamma') \vdash exp\ atexp \Rightarrow (con, v)} \tag{204}$$

$$\frac{(FE,\gamma) \vdash exp \Rightarrow en \qquad (FE,\gamma') \vdash atexp \Rightarrow v}{(FE,\gamma \cdot \gamma') \vdash exp\ atexp \Rightarrow (en, v)} \tag{205}$$

$$\frac{(FE,\gamma) \vdash exp \Rightarrow b \qquad (FE,\gamma') \vdash atexp \Rightarrow v \qquad \mathrm{APPLY}(b,v) = v'/p}{(FE,\gamma \cdot \gamma') \vdash exp\ atexp \Rightarrow v'/p} \tag{206}$$

$$\frac{\begin{array}{c} s_1,(FE,\gamma) \vdash exp \Rightarrow (match, M, VE), s_2 \qquad s_2,(FE,\gamma') \vdash atexp \Rightarrow v, s_3 \\ s_3 \oplus 0, M + \mathrm{Rec}\, VE,\ v \vdash match \Rightarrow v', s_4 \end{array}}{s_1,(FE,\gamma \cdot \gamma') \vdash exp\ atexp \Rightarrow v', s_4 \oplus (n \text{ of } s_3)} \tag{207}$$

$$\frac{\begin{array}{c} s_1,(FE,\gamma) \vdash exp \Rightarrow (match, M, VE), s_2 \qquad s_2,(FE,\gamma') \vdash atexp \Rightarrow v, s_3 \\ s_3 \oplus 0, M + \mathrm{Rec}\, VE,\ v \vdash match \Rightarrow \mathrm{FAIL}, s_4 \end{array}}{s_1,(FE,\gamma \cdot \gamma') \vdash exp\ atexp \Rightarrow [\texttt{Match}], s_4 \oplus (n \text{ of } s_3)} \tag{208}$$

$$\frac{M \vdash exp \Rightarrow v}{M \vdash exp\ :\ ty \Rightarrow v} \tag{209}$$

$$\frac{\dfrac{\gamma \succ \gamma' = (C,\tau) \cdot \gamma''}{s, FE, \gamma' \vdash exp_1^\bullet \ \equiv\ exp_2^\bullet \Rightarrow v}}{s,(FE,\gamma) \vdash exp_1^\bullet \ \texttt{==}\ exp_2^\bullet \Rightarrow v, s} \tag{210}$$

Comments:

(207),(208) To verificate *match* we have to reset the n of s component to 0, because the question mark interpretation in M indexes occurrences of question marks starting from 0; see also rule 223 and the definition of Dyn. Similarly, the n of s component of the final state is taken from s_3, because *match* is not a subphrase of *exp atexp*.

(209) The type expression *ty* is ignored. The type it describes may not be "accurate" (in an intuitive sense) anyway, because the explicit type variables in *exp* are subject to abstraction and instantiation. If we needed to obtain type information from *ty* here, we would have to extend the environment by a type variable environment, mapping type variables to types. But this is fortunately unnecessary since type information (coming from the static semantics) is provided by traces, see rule 341.

(210) Equality is type dependent. However, the premise requires that it is not dependent on the instantiation of type variables local to the expression — all instances should give the same result. Note that the state does not change, even if exp_1 and/or exp_2 change the state.

$$\frac{\begin{array}{c}\mathrm{Comp}(FE,s) = VE \qquad \gamma = (C,\tau)\cdot\gamma' \qquad \alpha^{(k)} \cap \mathrm{tyvars}(FE) = \emptyset \\ s^{\#\#}(C) + \mathrm{Stat}\,VE \vdash_{\mathrm{STAT}} atexp^{\bullet} \Rightarrow \tau', \emptyset, \gamma'' \qquad s^{\#\#}(\tau) = \tau' \\ \mathrm{Dyn}(s, FE + VE) \vdash_{\mathrm{DYN}} atexp^{\bullet} \Rightarrow v_{\mathrm{DYN}}, (\top, ens) \\ s, (FE + VE, \gamma\cdot\gamma'') \vdash (\texttt{fn}\ match^{\bullet})\ atexp^{\bullet} \Rightarrow \texttt{true}, s'\end{array}}{s, (FE, \forall\alpha^{(k)}.\gamma) \vdash \texttt{exists}\ match^{\bullet} \Rightarrow \texttt{true}, s} \tag{211}$$

$$\frac{\begin{array}{c}\mathrm{Comp}(FE,s) = VE \\ \left(\begin{array}{c}\gamma \succ \gamma_1 = (C,\tau)\cdot\gamma_2 \quad s^{\#\#}(C) + \mathrm{Stat}\,VE \vdash_{\mathrm{STAT}} atexp^{\bullet} \Rightarrow \tau', \emptyset, \gamma_3 \\ s^{\#\#}(\tau) = \tau' \quad \mathrm{Dyn}(s, FE + VE) \vdash_{\mathrm{DYN}} atexp^{\bullet} \Rightarrow v_{\mathrm{DYN}}, (\top, ens)\end{array}\right) \\ \exists s'.\ s, (FE + VE, \gamma_1\cdot\gamma_3) \vdash (\texttt{fn}\ match^{\bullet})\ atexp^{\bullet} \Rightarrow \texttt{false}, s'\end{array}}{s, (FE, \gamma) \vdash \texttt{exists}\ match^{\bullet} \Rightarrow \texttt{false}, s} \tag{212}$$

$$\frac{\begin{array}{c}\mathrm{Comp}(FE,s) = VE \\ \left(\begin{array}{c}\gamma \succ \gamma_1 = (C,\tau)\cdot\gamma_2 \quad s^{\#\#}(C) + \mathrm{Stat}\,VE \vdash_{\mathrm{STAT}} atexp^{\bullet} \Rightarrow \tau', \emptyset, \gamma_3 \\ s^{\#\#}(\tau) = \tau' \quad \mathrm{Dyn}(s, FE + VE) \vdash_{\mathrm{DYN}} atexp^{\bullet} \Rightarrow v_{\mathrm{DYN}}, (\top, ens)\end{array}\right) \\ \exists s'.\ s, (FE + VE, \gamma_1\cdot\gamma_3) \vdash (\texttt{fn}\ match^{\bullet})\ atexp^{\bullet} \Rightarrow \texttt{true}, s'\end{array}}{s, (FE, \gamma) \vdash \texttt{forall}\ match^{\bullet} \Rightarrow \texttt{true}, s} \tag{213}$$

$$\frac{\begin{array}{c}\mathrm{Comp}(FE,s) = VE \qquad \gamma = (C,\tau)\cdot\gamma' \qquad \alpha^{(k)} \cap \mathrm{tyvars}(FE) = \emptyset \\ s^{\#\#}(C) + \mathrm{Stat}\,VE \vdash_{\mathrm{STAT}} atexp^{\bullet} \Rightarrow \tau', \emptyset, \gamma'' \qquad s^{\#\#}(\tau) = \tau' \\ \mathrm{Dyn}(s, FE + VE) \vdash_{\mathrm{DYN}} atexp^{\bullet} \Rightarrow v_{\mathrm{DYN}}, (\top, ens) \\ s, (FE + VE, \gamma\cdot\gamma'') \vdash (\texttt{fn}\ match^{\bullet})\ atexp^{\bullet} \Rightarrow \texttt{false}, s'\end{array}}{s, (FE, \forall\alpha^{(k)}.\gamma) \vdash \texttt{forall}\ match^{\bullet} \Rightarrow \texttt{false}, s} \tag{214}$$

Comments:

(211)–(214) Verification of a quantified expression does not change the state. All state changes that happen during its verification are recovered. The purpose of $\mathrm{Comp}(FE, s)$ is to complete the environment to make all values accessible by syntactic means. In other words: quantification ranges over all expressible values, including even all values of abstract types in signatures, and not just over those values that are expressible at the particular point in the program where the quantified expression occurs.

Quantification ranges only over defined values (no packets), therefore the required evaluation to a value.

(211),(214) Requiring $\alpha^{(k)}$ to be distinct from the type variables in *FE* corresponds to instantiating the trace scheme with fresh type variables. The purpose of this requirement is to guarantee that the witness $atexp^\bullet$ serves as a witness for any instantiation of the trace scheme. $atexp^\bullet$ is an arbitrary (atomic) expression which is well-formed in context $s^{\#\#}(C)$ giving a type τ' that is the same type as $s^{\#\#}(\tau)$.

(212),(213) The premises of these rules contain a rule themselves, i.e. these are higher-order rules.

Notice that the quantifier rules are not complete in the sense that a quantified expression does not necessarily verificate to either `true` or `false`. The "missing case" occurs when we are not able to provide a witness for truth of an existentially quantified formula (resp. falsity of a universally quantified formula) which is as polymorphic as the *match*, but the existentially quantified formula is not `false` (resp. the universally quantified formula is not `true`) because on some values given by $atexp^\bullet$ the body of the formula does not verificate to `true` (resp. `false`) because it either verificates to `false` (resp. `true`), or raises an exception, or does not terminate. An important consequence of this is that a polymorphically quantified formula is taken to be undefined if it is true for type instances and false for others.

$$\frac{\mathrm{Dyn}(s, M) \vdash_{\mathrm{DYN}} exp^\bullet \Rightarrow v_{\mathrm{DYN}}/p_{\mathrm{DYN}}, (\top, ens)}{s, M \vdash exp^\bullet \texttt{ terminates} \Rightarrow \texttt{true}, s} \tag{215}$$

$$\frac{\mathrm{Dyn}(s, M) \vdash_{\mathrm{DYN}} exp^\bullet \Rightarrow v_{\mathrm{DYN}}/p_{\mathrm{DYN}}, (\bot, ens)}{s, M \vdash exp^\bullet \texttt{ terminates} \Rightarrow [\texttt{Abuse}], s} \tag{216}$$

$$\frac{\begin{array}{c} s_{\mathrm{DYN}}, E_{\mathrm{DYN}} = \mathrm{Dyn}(s, M) \\ \neg\exists v_{\mathrm{DYN}}/p_{\mathrm{DYN}}, s'_{\mathrm{DYN}}.\ s_{\mathrm{DYN}}, E_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} exp^\bullet \Rightarrow v_{\mathrm{DYN}}/p_{\mathrm{DYN}}, s'_{\mathrm{DYN}} \end{array}}{s, M \vdash exp^\bullet \texttt{ terminates} \Rightarrow \texttt{false}, s} \tag{217}$$

Comments:

(215) Raising an exception is treated as a terminating case.

Notice that the flag *sp* of the dynamic state must stay $\top$ to make the convergence predicate `true`. As any specification construct sets the flag to $\bot$, no such construct can occur during an evaluation that does not change the flag.

(216) If an evaluation terminates in the dynamic semantics but the flag *sp* is set to $\bot$, then the verification semantics might or might not derive a semantic value for the same evaluation. The packet [`Abuse`] indicates an abuse of the convergence predicate — it is not intended to be used for expressions containing specification constructs.

(217) The convergence predicate is verificated using the dynamic semantics. This is necessary to avoid paradoxes caused by the non-existence premise of this rule.

$$\frac{(FE, \gamma) \vdash exp \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash exp \texttt{ handle } match \Rightarrow v} \tag{218}$$

Comments:

(210)–(218) These rules are exempted from the exception convention.

$$\frac{(FE, \gamma) \vdash exp \Rightarrow [e] \qquad e \neq \texttt{Abuse} \qquad (FE, \gamma'), e \vdash match \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash exp \texttt{ handle } match \Rightarrow v} \tag{219}$$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow [e] \qquad e \neq \texttt{Abuse} \qquad (FE, \gamma'), e \vdash match \Rightarrow \text{FAIL}}{(FE, \gamma \cdot \gamma') \vdash exp \texttt{ handle } match \Rightarrow [e]} \tag{220}$$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow [\texttt{Abuse}]}{(FE, \gamma \cdot \gamma') \vdash exp \texttt{ handle } match \Rightarrow [\texttt{Abuse}]} \tag{221}$$

Comments:

(219)–(221) The exception `Abuse` is treated specially, similarly to `NoCode` in the dynamic semantics. We do not need special treatment for `NoCode` here, because it is never raised in the verification semantics.

$$\frac{(FE, \gamma) \vdash exp \Rightarrow e}{(FE, \tau \cdot \gamma) \vdash \texttt{raise } exp \Rightarrow [e]} \tag{222}$$

$$\frac{\begin{array}{c} n = n \text{ of } s \qquad M = ((E, (QIE, QIT)), (C, \tau) \cdot \gamma) \\ (match', n') = \text{Replace}(match, (QIE, QIT), n) \\ s^{\#}(C) \vdash_{\text{STAT}} match' \Rightarrow \tau', U, \gamma' \qquad s^{\#}(\tau) = \tau' \\ f = \lambda k.k - n \qquad QIE' = QIE \circ f \qquad QIT' = QIT \circ f \\ M' = ((E, (QIE', QIT')), \gamma) \end{array}}{s, M \vdash \texttt{fn } match \Rightarrow (match, M', \{\}), s \oplus n'} \tag{223}$$

Comments:

(223) The purpose of substituting all question marks in the premise is to avoid any ill-typed interpretations for the question marks when the closure is applied to an argument. The so-obtained phrase $match'$ is not used for the closure for technical reasons (traces and type interpretations are not quite right).

The difference between n' and n is the number of question marks occurring in $match$. When verificating a closure, we have to know at which number the labelling of question marks starts. The composition of the question mark interpretations with f makes it start from 0; this convention is used for closure application (rules 207 and 208) and for mapping verification values to dynamic values in the definition of Dyn.

The set of unguarded type variables U in $match'$ is left unspecified. If we wanted to consider QIE as textual substitution, we would need that all type variables in U are free in the context C.

Equality

$$\boxed{s, FE, \gamma \vdash exp_1 \ \equiv \ exp_2 \Rightarrow v}$$

$$\frac{\begin{array}{c} s,(FE,\gamma_1) \vdash exp_1 \Rightarrow v_1, s_1 \qquad s,(FE,\gamma_2) \vdash exp_2 \Rightarrow v_2, s_2 \\ s, FE, C, \tau \vdash v_1 \approx v_2 \Rightarrow v \end{array}}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow v} \tag{224}$$

$$\frac{\begin{array}{c} s,(FE,\gamma_1) \vdash exp_1 \Rightarrow [e_1], s_1 \qquad s,(FE,\gamma_2) \vdash exp_2 \Rightarrow [e_2], s_2 \\ s, FE, C, \texttt{exn} \vdash e_1 \approx e_2 \Rightarrow v \end{array}}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow v} \tag{225}$$

$$\frac{s,(FE,\gamma_1) \vdash exp_1 \Rightarrow v, s_1 \qquad s,(FE,\gamma_2) \vdash exp_2 \Rightarrow p, s_2}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow \texttt{false}} \tag{226}$$

$$\frac{s,(FE,\gamma_1) \vdash exp_1 \Rightarrow p, s_1 \qquad s,(FE,\gamma_2) \vdash exp_2 \Rightarrow v, s_2}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow \texttt{false}} \tag{227}$$

$$\frac{\begin{array}{c} s,(FE,\gamma_1) \vdash exp_1 \ \texttt{terminates} \Rightarrow \texttt{false}, s \\ s,(FE,\gamma_2) \vdash exp_2 \ \texttt{terminates} \Rightarrow \texttt{false}, s \end{array}}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow \texttt{true}} \tag{228}$$

$$\frac{\begin{array}{c} s,(FE,\gamma_1) \vdash exp_1 \Rightarrow v/p, s_1 \\ s,(FE,\gamma_2) \vdash exp_2 \ \texttt{terminates} \Rightarrow \texttt{false}, s \end{array}}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow \texttt{false}} \tag{229}$$

$$\frac{\begin{array}{c} s,(FE,\gamma_1) \vdash exp_1 \ \texttt{terminates} \Rightarrow \texttt{false}, s \\ s,(FE,\gamma_2) \vdash exp_2 \Rightarrow v/p, s \end{array}}{s, FE, (C,\tau)\cdot\gamma_1\cdot\gamma_2 \vdash exp_1 \ \equiv \ exp_2 \Rightarrow \texttt{false}} \tag{230}$$

Comments:

(224),(225) The last sentence in the premise of both rules refers to the comparison rules (for comparing *values*), 231 to 232.

(225) The type τ is arbitrary (and does not contribute to the result) when comparing packets, because raising an exception gives an arbitrary type.

(228)–(230) Notice that the sentence $s, M \vdash \mathit{atexp}\ \texttt{terminates} \Rightarrow \texttt{false}, s$ (see the premises of these rules) ensures that *atexp* has no semantic value, i.e. the rules do not overlap. This is guaranteed by the rules for the convergence predicate, see rule 217, and the relationship between the dynamic and verification semantics, see Section 8.7.

The predicate $\equiv$ is a partial congruence relation. It is partial because it is undefined if in $exp_1 \equiv exp_2$ one of the exp_i has no semantic value *and* the expression exp_i `terminates` evaluates to [`Abuse`]. This can only happen — see rule 216 — if the dynamic evaluation of exp_i terminates *while* setting the flag sp to $\bot$, indicating that a specification construct has been encountered.

Comparisons

$$\boxed{s, FE, C, \tau \vdash v \approx v' \Rightarrow v''}$$

$$\frac{\begin{array}{c} VE = \mathrm{Comp}(FE, s) \qquad id \notin \mathrm{Dom}(VE \text{ of } (FE + VE)) \\ VE_1 = VE + \{id \mapsto (v, \tau)\} \qquad VE_2 = VE + \{id \mapsto (v', \tau)\} \\ s^{\#\#}(C) + \mathrm{Stat}\, VE_1 \vdash_{\mathrm{STAT}} exp \Rightarrow \texttt{bool}, \emptyset, \gamma \\ s, (FE + VE_1, \gamma) \vdash exp \Rightarrow v_1, s' \qquad s, (FE + VE_2, \gamma) \vdash exp \Rightarrow v_2, s'' \\ \mathrm{APPLY}(\texttt{=}, \{\texttt{1} \mapsto v_1, \texttt{2} \mapsto v_2\}) = \texttt{false} \end{array}}{s, FE, C, \tau \vdash v \approx v' \Rightarrow \texttt{false}} \tag{231}$$

$$\frac{\begin{array}{c} VE = \mathrm{Comp}(FE, s) \qquad id \notin \mathrm{Dom}(VE \text{ of } (FE + VE)) \\ VE_1 = VE + \{id \mapsto (v, \tau)\} \qquad VE_2 = VE + \{id \mapsto (v', \tau)\} \\ \left(\begin{array}{c} s^{\#\#}(C) + \mathrm{Stat}\, VE_1 \vdash_{\mathrm{STAT}} exp \Rightarrow \texttt{bool}, \emptyset, \gamma \\ \dfrac{s, (FE + VE_1, \gamma) \vdash exp \Rightarrow v_1, s' \qquad s, (FE + VE_2, \gamma) \vdash exp \Rightarrow v_2, s''}{\mathrm{APPLY}(\texttt{=}, \{\texttt{1} \mapsto v_1, \texttt{2} \mapsto v_2\}) = \texttt{true}} \end{array} \right) \end{array}}{s, FE, C, \tau \vdash v \approx v' \Rightarrow \texttt{true}} \tag{232}$$

Comments:

(231),(232) Values are considered equal if and only if they cannot be distinguished by expressions of type `bool`. $v \approx v'$ tests that the values are indistinguishable in any context, not only the current one — hence the use of $\mathrm{Comp}(FE, s)$ to complete the environment similarly as for quantification (rules 211–214). To compare the values, we further extend the environment by binding a fresh identifier *id* to v resp. v', and then check if there is an expression of type `bool` that distinguishes these two environments.

Since Stat VE_1 = Stat VE_2, only one premise ensuring elaboration is required.

(224)–(232) The exception convention does not apply to these rules.

Matches

$$\boxed{M, v \vdash match \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{(FE, \gamma), v \vdash mrule \Rightarrow v'}{(FE, \gamma\langle\cdot\gamma'\rangle), v \vdash mrule\ \langle\ \texttt{|}\ match\rangle \Rightarrow v'} \tag{233}$$

$$\frac{M, v \vdash mrule \Rightarrow \text{FAIL}}{M, v \vdash mrule \Rightarrow \text{FAIL}} \tag{234}$$

$$\frac{(FE, \gamma), v \vdash mrule \Rightarrow \text{FAIL} \qquad (FE, \gamma'), v \vdash match \Rightarrow v'/\text{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash mrule\ \texttt{|}\ match \Rightarrow v'/\text{FAIL}} \tag{235}$$

Match Rules

$$\boxed{M, v \vdash mrule \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{(FE, \gamma), v \vdash pat \Rightarrow VE \qquad (FE + VE, \gamma') \vdash exp \Rightarrow v'}{(FE, \gamma \cdot \gamma'), v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow v'} \tag{236}$$

$$\frac{(FE, \gamma), v \vdash pat \Rightarrow \text{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow \text{FAIL}} \tag{237}$$

Declarations

$$\boxed{M \vdash dec \Rightarrow E/p}$$

$$\frac{\gamma \notin \text{TraceScheme} \qquad \alpha^{(k)} \cap \text{tyvars}(FE) = \emptyset \qquad (FE, \gamma) \vdash valbind \Rightarrow VE \qquad VE' = \text{Clos}_{FE} VE}{(FE, \forall\alpha^{(k)}.\gamma) \vdash \texttt{val}\ valbind \Rightarrow VE' \text{ in Env}} \tag{238}$$

$$\frac{s_1, (FE, \gamma) \vdash typbind \Rightarrow \{\}, s_2}{s_1, (FE, TE \cdot \gamma) \vdash \texttt{type}\ typbind \Rightarrow s_2^{\#}(TE) \text{ in Env},\ s_2} \tag{239}$$

$$\frac{s_1, (FE, \gamma) \vdash typbind \Rightarrow \{\}, s_2}{s_1, (FE, TE \cdot \gamma) \vdash \texttt{eqtype}\ typbind \Rightarrow s_2^{\#}(TE) \text{ in Env},\ s_2} \tag{240}$$

$$\frac{CT = s^{\#}(\text{Graph}(VE \text{ of } E_{\text{STAT}})) \qquad s, (FE, \gamma) \vdash datbind \Rightarrow VE, s}{\begin{array}{c} s, (FE, E_{\text{STAT}} \cdot \gamma) \vdash \texttt{datatype}\ datbind \Rightarrow \\ s^{\#}(VE, TE \text{ of } E_{\text{STAT}}) \text{ in Env},\ s + CT \end{array}} \tag{241}$$

$$\frac{\begin{array}{c} CT = s^{\#}(\text{Graph}(VE_{\text{STAT}})) \\ s, (FE, \gamma) \vdash datbind \Rightarrow VE, s \qquad s + CT, (FE + VE, \gamma') \vdash dec \Rightarrow E, s_1 \\ E = \varphi_{\text{Ty}}(E') \qquad \text{tynames}(E') \cap \text{Yield}(\varphi_{\text{Ty}}) = \emptyset \qquad s_2 = s_1 + \varphi_{\text{Ty}} \end{array}}{s, (FE, (VE_{\text{STAT}}, \varphi_{\text{Ty}}) \cdot \gamma \cdot \gamma') \vdash \texttt{abstype}\ datbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow E', s_2} \tag{242}$$

$$\frac{s, M \vdash exbind \Rightarrow VE, s'}{s, M \vdash \texttt{exception}\ exbind \Rightarrow s^{\#}(VE)\ \text{in Env}, s'} \tag{243}$$

$$\frac{(FE, \gamma) \vdash dec_1 \Rightarrow E \qquad (FE + E, \gamma') \vdash dec_2 \Rightarrow E'}{(FE, \gamma \cdot \gamma') \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E'} \tag{244}$$

$$\frac{M(longstrid_1) = (E_1, m_1) \quad \cdots \quad M(longstrid_n) = (E_n, m_n)}{s, M \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n, s} \tag{245}$$

$$\frac{}{M \vdash \qquad \Rightarrow \{\}\ \text{in Env}} \tag{246}$$

$$\frac{(FE, \gamma) \vdash dec_1 \Rightarrow E \qquad (FE + E, \gamma') \vdash dec_2 \Rightarrow E'}{(FE, \gamma \cdot \gamma') \vdash dec_1\ \langle ; \rangle\ dec_2 \Rightarrow E + E'} \tag{247}$$

Comments:

(238) Dropping the quantifier is a particular way of instantiating the trace scheme $\forall \alpha^{(k)}.\gamma$. Other instantiations might lead to different variable environments, but only in pathological cases involving type-dependent specification constructs (e.g. quantification) and type assertions with explicit type variables to prevent the specification constructs from binding these type variables.

(241),(242) Remember that Graph is the graph of a finite map and that the component CT of the state is used for comparing values and for quantification.

The state s does not change when verificating a *datbind*, i.e. the requirement that no state change takes place does not restrict the applicability of the rule.

(242) The type realisation φ_{Ty} maps type names of abstract types to their implementing types, see rule 20 and the definition of $\mathrm{Abs_C}$. We apply it "backwards" to the environment E to obtain an environment E' in which the implementing types ($\mathrm{Yield}(\varphi_{\mathrm{Ty}})$) have been replaced by the corresponding abstract type names. Such an E' always exist because φ_{Ty} maps type names to type names, and it is uniquely determined because φ_{Ty} is injective.

Value Bindings

$\boxed{M \vdash valbind \Rightarrow VE/p}$

$$\frac{(FE, \gamma') \vdash exp \Rightarrow v \qquad (FE, \gamma), v \vdash pat \Rightarrow VE \qquad \langle (FE, \gamma'') \vdash valbind \Rightarrow VE' \rangle}{(FE, \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle) \vdash pat\ \texttt{=}\ exp\ \langle \texttt{and}\ valbind \rangle \Rightarrow VE\ \langle + VE' \rangle} \tag{248}$$

$$\frac{(FE, \gamma') \vdash exp \Rightarrow v \qquad (FE, \gamma), v \vdash pat \Rightarrow \mathrm{FAIL}}{(FE, \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle) \vdash pat\ \texttt{=}\ exp\ \langle \texttt{and}\ valbind \rangle \Rightarrow [\texttt{Bind}]} \tag{249}$$

$$\frac{M \vdash valbind \Rightarrow VE}{M \vdash \texttt{rec}\ valbind \Rightarrow \mathrm{Rec}\, VE} \tag{250}$$

Comments:

(248) In a value binding, verification order differs from elaboration order. This is the reason why the *first* verification (*exp*) is done using the *second* trace and vice versa. Although in general a difference between verification order and syntactic appearance may cause problems w.r.t. the replacement of question marks, in this particular case everything works smoothly, since question marks do not occur in patterns.

Type Bindings

$\boxed{M \vdash typbind \Rightarrow \{\}}$

$$\frac{\langle (FE, \gamma) \vdash typbind \Rightarrow \{\} \rangle}{(FE, \epsilon\langle\cdot\gamma\rangle) \vdash tyvarseq\ tycon\ \texttt{=}\ ty\ \langle \texttt{and}\ typbind \rangle \Rightarrow \{\}} \tag{251}$$

$$\frac{\begin{array}{c} n = n \text{ of } s \qquad (QIT \text{ of } FE)(n) = ty \qquad s^{\#}(C) \vdash_{\text{STAT}} ty \Rightarrow \tau \\ tyvarseq = \alpha^{(k)} \qquad \text{tyvars}(ty) \subseteq \alpha^{(k)} \qquad TI = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \\ s' = (s + TI) \oplus (n+1) \qquad \langle s', (FE, \gamma) \vdash typbind \Rightarrow \{\}, s'' \rangle \end{array}}{s, (FE, (C, t)\langle\cdot\gamma\rangle) \vdash tyvarseq\ tycon\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ typbind \rangle \Rightarrow \{\}, s'\langle' \rangle} \tag{252}$$

Comments:

(252) The question mark interpretation QIT maps occurrences of ? to type expressions. These type expressions can be built from the type constructors available so far, i.e. they have to elaborate at the position in the program at which the ? occurs.

Data Type Bindings

$\boxed{M \vdash datbind \Rightarrow VE}$

$$\frac{(FE, \gamma) \vdash conbind \Rightarrow VE \qquad \langle (FE, \gamma') \vdash datbind \Rightarrow VE' \rangle}{(FE, \gamma\langle\cdot\gamma'\rangle) \vdash tyvarseq\ tycon\ \texttt{=}\ conbind\ \langle \texttt{and}\ datbind \rangle \Rightarrow VE\ \langle + VE' \rangle} \tag{253}$$

Constructor Bindings

$\boxed{M \vdash conbind \Rightarrow VE}$

$$\frac{con = id^{\mathsf{C}} \qquad \langle (FE, \gamma) \vdash conbind \Rightarrow VE \rangle}{(FE, \tau\langle\cdot\gamma\rangle) \vdash con\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow \{id \mapsto \text{Clos}(con, \tau)\}\ \langle + VE \rangle} \tag{254}$$

$$\frac{con = id^{\mathsf{C}} \qquad \langle (FE, \gamma) \vdash conbind \Rightarrow VE \rangle}{(FE, \tau\langle\cdot\gamma\rangle) \vdash con\ \texttt{of}\ ty\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow \{id \mapsto \text{Clos}(con, \tau)\}\ \langle + VE \rangle} \tag{255}$$

Exception Bindings

$\boxed{M \vdash exbind \Rightarrow VE}$

$$\frac{\begin{array}{c} excon = id^{\mathsf{e}} \qquad en = \min(\text{ExName} \setminus (ens \text{ of } s)) \\ s' = s + (\{en\} \text{ in ExNameSet},\ \{(en,\ \texttt{exn})\} \text{ in ValTemp}) \\ \langle s', (FE, \gamma) \vdash exbind \Rightarrow VE, s'' \rangle \end{array}}{\begin{array}{l} s, (FE, \epsilon\langle\cdot\gamma\rangle) \vdash excon\ \langle \texttt{and}\ exbind\rangle \Rightarrow \\ \qquad \{id \mapsto (en, \texttt{exn})\}\ \langle + VE\rangle,\ s'\langle'\rangle \end{array}} \tag{256}$$

$$\frac{\begin{array}{c} excon = id^{\mathsf{e}} \qquad en = \min(\text{ExName} \setminus (ens \text{ of } s)) \qquad \tau' = s^{\#}(\tau) \to \texttt{exn} \\ s' = s + (\{en\} \text{ in ExNameSet},\ \{(en, \tau')\} \text{ in ValTemp}) \\ \langle s', (FE, \gamma) \vdash exbind \Rightarrow VE, s'' \rangle \end{array}}{\begin{array}{l} s, (FE, \tau\langle\cdot\gamma\rangle) \vdash excon\ \texttt{of}\ ty\ \langle \texttt{and}\ exbind\rangle \Rightarrow \\ \qquad \{id \mapsto (en, \tau')\}\ \langle + VE\rangle,\ s'\langle'\rangle \end{array}} \tag{257}$$

$$\frac{excon = id^{\mathsf{e}} \qquad FE(longexcon) = (en, \tau) \qquad \langle (FE, \gamma) \vdash exbind \Rightarrow VE\rangle}{\begin{array}{l} (FE, \epsilon\langle\cdot\gamma\rangle) \vdash excon\ \texttt{=}\ longexcon\ \langle \texttt{and}\ exbind\rangle \Rightarrow \\ \qquad \{id \mapsto (en, \tau)\}\langle + VE\rangle \end{array}} \tag{258}$$

Comments:

(256),(257) The use of min ensures that *excon* is uniquely determined, which is needed to make `==` reflexive, example:

```
val en = fn {} => let exception A in A end;
```

Successful verification of `en{} == en{}` requires a fixed result, see rule 210. Both occurrences of `en{}` will be verificated in the *same* state (see rule 224) and the deterministic choice of fresh exception names ensures that the same name is chosen in both cases.

Atomic Patterns

$\boxed{M, v \vdash atpat \Rightarrow VE/\text{FAIL}}$

$$\frac{}{M, v \vdash _ \Rightarrow \{\}} \tag{259}$$

$$\frac{v = \text{val}(scon)}{M, v \vdash scon \Rightarrow \{\}} \tag{260}$$

$$\frac{v \neq \text{val}(scon)}{M, v \vdash scon \Rightarrow \text{FAIL}} \tag{261}$$

$$\frac{var = id^{\mathsf{v}}}{s, (FE, \tau), v \vdash var \Rightarrow \{id \mapsto (v, s^{\#}(\tau))\}, s} \tag{262}$$

$$\frac{M(longcon) \succ (con, \tau) \qquad v = con}{M, v \vdash longcon \Rightarrow \{\}} \tag{263}$$

$$\frac{M(longcon) \succ (con, \tau) \qquad v \neq con}{M, v \vdash longcon \Rightarrow \text{FAIL}} \tag{264}$$

$$\frac{M(longexcon) = (v, \tau)}{M, v \vdash longexcon \Rightarrow \{\}} \tag{265}$$

$$\frac{M(longexcon) = (v', \tau) \qquad v' \neq v}{M, v \vdash longexcon \Rightarrow \text{FAIL}} \tag{266}$$

$$\frac{v = \{\}\langle +r\rangle \text{ in Val} \qquad \langle (FE, \gamma), r \vdash patrow \Rightarrow VE/\text{FAIL}\rangle}{(FE, \epsilon\langle\cdot\gamma\rangle), v \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow \{\}\langle +VE/\text{FAIL}\rangle} \tag{267}$$

$$\frac{M, v \vdash pat \Rightarrow VE/\text{FAIL}}{M, v \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow VE/\text{FAIL}} \tag{268}$$

Pattern Rows

$$\boxed{M, r \vdash patrow \Rightarrow VE/\text{FAIL}}$$

$$\frac{}{M, r \vdash \ldots \Rightarrow \{\}} \tag{269}$$

$$\frac{(FE, \gamma), r(lab) \vdash pat \Rightarrow \text{FAIL}}{(FE, \gamma\langle\cdot\gamma'\rangle), r \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow\rangle \Rightarrow \text{FAIL}} \tag{270}$$

$$\frac{(FE, \gamma), r(lab) \vdash pat \Rightarrow VE \qquad \langle (FE, \gamma'), r \vdash patrow \Rightarrow VE'/\text{FAIL}\rangle}{(FE, \gamma\langle\cdot\gamma'\rangle), r \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow\rangle \Rightarrow VE\langle +\ VE'/\text{FAIL}\rangle} \tag{271}$$

Patterns

$$\boxed{M, v \vdash pat \Rightarrow VE/\text{FAIL}}$$

$$\frac{M, v \vdash atpat \Rightarrow VE/\text{FAIL}}{M, v \vdash atpat \Rightarrow VE/\text{FAIL}} \tag{272}$$

$$\frac{\begin{array}{c} FE(longcon) \succ (con, \tau') \qquad v = (con, v') \\ (FE, \gamma), v' \vdash atpat \Rightarrow VE/\text{FAIL} \end{array}}{(FE, \tau \cdot \gamma), v \vdash longcon\ atpat \Rightarrow VE/\text{FAIL}} \tag{273}$$

$$\frac{M(longcon) \succ (con, \tau) \qquad v \notin \{con\} \times \text{Val}}{M, v \vdash longcon\ atpat \Rightarrow \text{FAIL}} \tag{274}$$

$$\frac{\begin{array}{c} M(longexcon) = (en, \tau) \qquad v = (en, v') \\ M, v' \vdash atpat \Rightarrow VE/\text{FAIL} \end{array}}{M, v \vdash longexcon\ atpat \Rightarrow VE/\text{FAIL}} \tag{275}$$

$$\frac{M(longexcon) = (en, \tau) \qquad v \notin \{en\} \times \mathrm{Val}}{M, v \vdash longexcon\ atpat \Rightarrow \mathrm{FAIL}} \tag{276}$$

$$\frac{M, v \vdash pat \Rightarrow VE/\mathrm{FAIL}}{M, v \vdash pat\ :\ ty \Rightarrow VE/\mathrm{FAIL}} \tag{277}$$

$$\frac{(FE, \gamma), v \vdash var \Rightarrow VE \qquad (FE, \gamma'), v \vdash pat \Rightarrow VE'/\mathrm{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash var\langle :\ ty\rangle\ \texttt{as}\ pat \Rightarrow VE + VE'/\mathrm{FAIL}} \tag{278}$$

Type Expressions and Type-expression Rows

There are no primitive sentence forms for these phrase classes; although type expressions are present in the verification semantics for the Core, they are either disregarded (e.g. rule 209) or interpreted via the static semantics (e.g. rule 252).

9 Verification Semantics for Modules

9.1 Compound Objects

The compound objects for the Modules verification semantics, extra to those for the Core verification semantics, are shown in Figure 18. For each semantic class A there is a class $\mathrm{Set}(A) = \wp(\mathrm{State} \times A)$; each $\mathcal{A} \in \mathrm{Set}(A)$ is a (possibly infinite) set of pairs (s, a), $s \in \mathrm{State}$, $a \in A$. We use the same conventions as in the verification semantics for the Core to refer to semantic objects of the static and dynamic semantics.

$$
\begin{array}{rcl}
(B, \gamma, axdesc) & \in & \mathrm{BasicAx} = \mathrm{Basis} \times \mathrm{Trace} \times \mathrm{AxDesc} \\
(N, B)(I, A) & \in & \mathrm{ExistAx} = \mathrm{NameSet} \times \mathrm{Basis} \times (\mathrm{Int} \times \mathrm{GenAx}) \\
A & \in & \mathrm{GenAx} = \mathrm{BasicAx} \uplus \mathrm{ExistAx} \uplus \\
 & & \quad (\mathrm{GenAx} \times \mathrm{GenAx}) \uplus \mathrm{Bit} \\
(IE, TE, VE_{\mathrm{STAT}}, A) \text{ or } I & \in & \mathrm{Int} = \mathrm{IntEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv}_{\mathrm{STAT}} \times \mathrm{GenAx} \\
(I, m) \text{ or } IS & \in & \mathrm{IntStr} = \mathrm{Int} \times \mathrm{StrName} \\
IE & \in & \mathrm{IntEnv} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{IntStr} \\
(N)IS \text{ or } \Sigma & \in & \mathrm{Sig} = \mathrm{NameSet} \times \mathrm{IntStr} \\
(strid, B, (N)(IS, \Sigma)) & \in & \mathrm{FunctorClosure} = \\
 & & \quad \mathrm{StrId} \times \mathrm{Basis} \times \mathrm{NameSet} \times \mathrm{IntStr} \times \mathrm{Sig} \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \xrightarrow{\mathrm{fin}} \mathrm{Sig} \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{FunctorClosure} \\
(N, F, G, E) \text{ or } B & \in & \mathrm{Basis} = \mathrm{NameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
(N, F, G, IE, E) \text{ or } IB & \in & \mathrm{IntBasis} = \mathrm{NameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \\
 & & \quad \mathrm{IntEnv} \times \mathrm{Env} \\
\mathcal{S} & \in & \mathrm{Set}(\mathrm{Str}) \\
\mathcal{R} & \in & \mathrm{Set}(\mathrm{Rea} \times \mathrm{Str}) \\
\mathcal{E} & \in & \mathrm{Set}(\mathrm{Env}) \\
\mathcal{SE} & \in & \mathrm{Set}(\mathrm{StrEnv}) \\
\mathcal{B} & \in & \mathrm{Set}(\mathrm{Basis})
\end{array}
$$

Figure 18: Further Compound Semantic Objects

An *interface* $I \in \mathrm{Int}$ represents a "view" of a structure. Specifications will verificate to interfaces; moreover, during the verification of a specification or signature expression, structures (to which a specification may refer via "`open`") are represented only by their interfaces. A *signature* $\Sigma \in \mathrm{Sig}$ has the form $(N)(I, m)$. Signatures are similar to signatures in the static semantics, as they can be imposed on structures, requiring certain checks and determining a matching realisation, but they are also similar to interfaces in the dynamic semantics, as a signature includes a rudimentary environment with only static information.

A *functor closure* $(strid, B, (N)(IS, \Sigma))$ has three components. It contains the structure identifier *strid* of its argument structure and the basis B in which the functor binding appeared for the same reasons that functor closures in the dynamic semantics do. However, it does not contain the body of the functor *strexp*, because the semantics of a functor only depends on its interface, not on its implementation. The third component of a functor closure corresponds to a static functor signature, the only difference being that interface structures contain more information than static structures.

To extract an interface from an environment we define the operation Inter : Env $\to$ Int as follows:

$$\begin{array}{rcl} \text{Inter}(SE, TE, VE) & = & (IE, TE, \text{Stat}\, VE, \bot) \\ \text{where } IE & = & \{strid \mapsto (\text{Inter}\, E, m)\ ;\ SE(strid) = (E, m)\} \end{array}$$

An *interface structure* IS is an instance of a signature $\Sigma = (N)IS'$, written $\Sigma \geq IS$, if there exists a realisation φ such that $\varphi(IS') = IS$ and $\text{Supp}\, \varphi \subseteq N$. We write $\Sigma \geq_\varphi IS$ if we want to make φ explicit. This is analogous to signature instantiation in the static semantics, see section 5.9.

An *interface basis* $IB = (N, F, G, IE, E)$ is derived from a basis for verificating signature expressions and specifications. An interface basis in the verification semantics contains more components than an interface basis in the dynamic semantics — the extra components are mainly for the interpretation of axioms.

The function Inter is extended to create an interface basis from a basis B as follows:

$$\text{Inter}(N, F, G, E) \ = \ (N, F, G, IE \text{ of } (\text{Inter}\, E), E)$$

Specifications may hide components of an environment by reusing identifiers. This affects the verification of axiom specifications. We define the environment $E \setminus I$ as the restriction of E to identifiers not specified in I. More precisely, given an environment $E = (SE, TE, VE)$ and an interface I, we define the environment $E \setminus I$ to be (SE', TE', VE'), which is the same as E except that the domains are restricted as follows:

$$\begin{array}{rcl} \text{Dom}\, SE' & = & \text{Dom}\, SE \setminus \text{Dom}(IE \text{ of } I) \\ \text{Dom}\, TE' & = & \text{Dom}\, TE \setminus \text{Dom}(TE \text{ of } I) \\ \text{Dom}\, VE' & = & \text{Dom}\, VE \setminus \text{Dom}(VE_{\text{STAT}} \text{ of } I) \end{array}$$

9.2 Generalised Axioms

A *generalised axiom* $A \in \text{GenAx}$ is a "mobile" axiom, capable of being interpreted in different environments. Generalised axioms arise when verificating axioms in signature expressions, and checking that a structure matches a signature involves interpreting generalised axioms from the signature in the environment corresponding to the structure. The basis B in a generalised axiom $(B, \gamma, axdesc)$ has the

same purpose as the environment in a closure — because the axiom is mobile, it has to carry the basis of its original occurrence. The trace γ is (originally) the one obtained from the static analysis of *axdesc*; it is modified in the course of matching a structure against a signature, which involves applying a realisation to an interface. The interface I in a generalised axiom $(N, B)(I, A)$ can be viewed as a kind of existential quantification: some auxiliary structure matching the signature $(N)(I, m)$ (for an appropriate m) has to be found, which builds on components in basis B, such that A holds. A pair of generalised axioms (A_1, A_2) can be understood as their logical conjunction. The generalised axiom $\top$ is always satisfied. It serves as a default value for GenAx — for example, (VE_{STAT} in Int) stands for $(\{\}, \{\}, VE_{\text{STAT}}, \top)$. The generalised axiom $\bot$ is never satisfied.

9.3 Combining Interfaces

The sequential composition of specifications $spec_1 \langle ; \rangle spec_2$ is a bit delicate, as $spec_2$ can use and/or overwrite identifiers specified in $spec_1$. The corresponding semantic operation on interfaces has to reflect this. For a given interface basis IB we define the *sequential combination* of the interfaces I_1 and I_2 as the interface $I_1 \oplus_{IB} I_2$, where $\oplus_{IB}$ is defined as follows. Let $I_k = (IE_k, TE_k, VE^k_{\text{STAT}}, A_k)$, $k \in \{1, 2\}$. We write B of IB for the basis obtained from IB by removing the IE component. Below we use the notation $A \upharpoonright A'$ (for arbitrary finite maps A and A' of the same type) to denote the finite map A'' with $\operatorname{Dom} A'' = \operatorname{Dom} A \cap \operatorname{Dom} A'$ and for all $x \in \operatorname{Dom} A''$, $A''(x) = A(x)$.

$$
\begin{array}{rcl}
I_1 -_N I_2 & = & (N')((IE_1 \upharpoonright IE_2, TE_1 \upharpoonright TE_2, VE^1_{\text{STAT}} \upharpoonright VE^2_{\text{STAT}}, \top), m) \\
 & & \text{where} \left\{ \begin{array}{rcl} N' & = & \{m\} \cup \text{names } I_2 \setminus \text{names } I_1 \setminus N \\ m & \notin & N \cup \text{names } I_1 \cup \text{names } I_2 \end{array} \right. \\
I_1 \oplus_{IB} I_2 & = & (IE_1 + IE_2, TE_1 + TE_2, VE^1_{\text{STAT}} + VE^2_{\text{STAT}}, \\
 & & \qquad ((N, B \text{ of } IB)(I, A_1), A_2)) \\
 & & \text{where } (N)I = I_1 -_{N \text{ of } IB} I_2
\end{array}
$$

The operation $I_1 -_N I_2$ is purely auxiliary; it maps two interfaces and a name set to a signature. The idea is that this signature represents the hidden part (of I_1) when sequentially composing I_1 and I_2. Matching a structure against $I_1 \oplus_{IB} I_2$ also requires finding a structure matching the hidden part.

Notice that the operator $\oplus_{IB}$ is not associative, because $(I_1 \oplus_{IB} I_2) \oplus_{IB} I_3$ and $I_1 \oplus_{IB} (I_2 \oplus_{IB} I_3)$ differ in their last component, the generalised axiom. But we claim that the so-obtained generalised axioms A and A' are semantically equivalent in the sense that a sentence of the form $s, E \vdash A \Rightarrow \{\}$ (see rules 287 to 290 below) can be derived iff the sentence $s, E \vdash A' \Rightarrow \{\}$ can be derived as well.

9.4 Extracting Objects of the Static Semantics

We extend the family of functions Stat defined in Section 8.6 to semantic objects of the module semantics as follows:

$$\begin{aligned}
\mathrm{Stat}(N,F,G,E) &= (N, \mathrm{Stat}\,F, \mathrm{Stat}\circ G, \mathrm{Stat}\,E)\\
\mathrm{Stat}(N,F,G,IE,E) &= (N, \mathrm{Stat}\,F, \mathrm{Stat}\circ G, \mathrm{Stat}\,E + (\mathrm{Stat}\circ IE))\\
\mathrm{Stat}\,F &= \{\mathit{funid} \mapsto (N)(\mathrm{Stat}\,\mathit{IS}, \mathrm{Stat}\,\Sigma)\ ;\\
&\qquad F(\mathit{funid}) = (\mathit{strid}, B, (N)(\mathit{IS}, \Sigma))\}\\
\mathrm{Stat}((N)\mathit{IS}) &= (N)\,\mathrm{Stat}\,\mathit{IS}\\
\mathrm{Stat}(I,m) &= (m, \mathrm{Stat}\,I)\\
\mathrm{Stat}(\mathit{IE},\mathit{TE},\mathit{VE}_{\mathrm{STAT}},A) &= (\mathrm{Stat}\circ\mathit{IE}, \mathit{TE}, \mathit{VE}_{\mathrm{STAT}})
\end{aligned}$$

Interfaces and environments are both mapped to static environments; similarly, bases and interface bases are both mapped to static bases.

9.5 Sets

The verification of a core phrase in a given state and environment typically leads to a pair consisting of some semantic object (value, environment, etc.) and a new state. For the verification semantics for Modules we typically have sets of such pairs as verification results.

There are two major reasons for this: first, we interpret a structure binding

```
structure S: SIG = T
```

not as a binding of the identifier `S` to (some restriction of) the structure `T`, but as the binding of `S` to *any* structure matching the signature `SIG`. In other words, we abstract from the concrete structure `T`.

The second reason is the presence of `?` in core phrases. The verification semantics for the Core operates with a *given* interpretation of question marks. The verification semantics for Modules enumerates all question mark interpretations that lead to successful verifications and collects the results, see rule 299.

9.6 Inference Rules

There is no state convention and no exception convention for the verification semantics for Modules: states are always made explicit.

In contrast to most other parts of the semantics, we have not only sentences of the form $A \vdash \mathit{phrase} \Rightarrow A'$ where *phrase* is a syntactic object, but also a variety of sentence forms (not involving syntax) that determine whether a given structure matches a signature.

The convention for referring to sentences of the static semantics is the same as in the Core verification semantics.

Satisfying Signatures

$$\boxed{s, \Sigma \vdash S \Rightarrow S', \varphi}$$

Sentences of this form can be read as: in state s, structure S successfully matches signature Σ via realisation φ, returning structure S'.

$$\frac{\mathrm{Supp}\,\varphi \subseteq N \qquad \varphi(IS) = (I, m) \qquad s, E \vdash I \Rightarrow E'}{s, (N)IS \vdash (E, m) \Rightarrow (E', m), \varphi} \tag{279}$$

Restricting Environments

$$\boxed{s, E \vdash I \Rightarrow E'}$$

In the dynamic semantics, the effect of "cutting down" an environment E to an interface I, written $E \downarrow I$, was defined in Section 7.2. In the verification semantics the situation is a bit more difficult: an environment is not just cut down to an interface, it also has to satisfy its (generalised) axiom, and the types have to fit. We therefore express this operation formally in terms of rules.

$$\frac{\begin{array}{c} s, SE \vdash IE \Rightarrow SE' \qquad TE \vdash TE' \Rightarrow TE'' \qquad VE \vdash VE_{\mathrm{STAT}} \Rightarrow VE' \\ E' = (SE', TE'', VE') \qquad s, E' \vdash A \Rightarrow \{\} \end{array}}{s, (SE, TE, VE) \vdash (IE, TE', VE_{\mathrm{STAT}}, A) \Rightarrow E'} \tag{280}$$

Restricting Structure Environments

$$\boxed{s, SE \vdash IE \Rightarrow SE'}$$

$$\frac{\begin{array}{c} s, SE \vdash IE \Rightarrow SE' \qquad s, E \vdash I \Rightarrow E' \end{array}}{\begin{array}{l} s, SE + \{strid \mapsto (E, m)\} \vdash IE + \{strid \mapsto (I, m)\} \Rightarrow \\ \qquad SE' + \{strid \mapsto (E', m)\} \end{array}} \tag{281}$$

$$\frac{}{s, SE \vdash \{\} \Rightarrow \{\}} \tag{282}$$

Restricting Type Environments

$$\boxed{TE \vdash TE' \Rightarrow TE''}$$

$$\frac{\begin{array}{c} TE \vdash TE' \Rightarrow TE'' \qquad CE = CE' \vee CE' = \{\} \end{array}}{\begin{array}{l} TE + \{tycon \mapsto (\theta, CE)\} \vdash TE' + \{tycon \mapsto (\theta, CE')\} \Rightarrow \\ \qquad TE'' + \{tycon \mapsto (\theta, CE')\} \end{array}} \tag{283}$$

$$\frac{}{TE \vdash \{\} \Rightarrow \{\}} \tag{284}$$

Comments:

(283) Notice that the side-condition closely relates to the enrichment of type structures, see section 5.11.

Restricting Variable Environments $\boxed{VE \vdash VE_{\mathrm{STAT}} \Rightarrow VE'}$

$$\frac{VE \vdash VE_{\mathrm{STAT}} \Rightarrow VE' \qquad \forall\alpha^{(k)}.\tau \succ_{\vartheta} \forall\beta^{(l)}.\tau'}{\begin{array}{c} VE + \{id \mapsto \forall\alpha^{(k)}.(v,\tau)\} \vdash VE_{\mathrm{STAT}} + \{id \mapsto \forall\beta^{(l)}.\tau'\} \Rightarrow \\ VE' + \{id \mapsto \forall\beta^{(l)}.(\vartheta(v),\tau')\} \end{array}} \tag{285}$$

$$\frac{}{VE \vdash \{\} \Rightarrow \{\}} \tag{286}$$

Comments:

(285) There is a certain amount of arbitrariness in this rule, similarly as in rules 195 and 238 in the verification semantics for the Core. The arbitrariness is again in the possibility that v may contain free type variables which are not in τ, and thus not in the domain of ϑ.

Satisfying Generalised Axioms $\boxed{s, E \vdash A \Rightarrow \{\}}$

$$\frac{}{s, E \vdash \top \Rightarrow \{\}} \tag{287}$$

$$\frac{s, E \vdash A_1 \Rightarrow \{\} \qquad s, E \vdash A_2 \Rightarrow \{\}}{s, E \vdash (A_1, A_2) \Rightarrow \{\}} \tag{288}$$

$$\frac{s, B + E, \gamma \vdash axdesc \Rightarrow \{\}}{s, E \vdash (B, \gamma, axdesc) \Rightarrow \{\}} \tag{289}$$

$$\frac{\begin{array}{c} m \notin (N \text{ of } B \cup N) \qquad s, B, (N)(I,m) \vdash strexp^{\bullet} \Rightarrow \mathcal{R} \\ (s', (\varphi, (E', m))) \in \mathcal{R} \qquad s', E + E' \vdash \varphi(A) \Rightarrow \{\} \end{array}}{s, E \vdash (N, B)(I, A) \Rightarrow \{\}} \tag{290}$$

Comments:

(290) Notice that the rule has two implicit existential quantifiers: we have to find *some* structure expression such that *some* environment obtained from its verification satisfies the axiom.

There is no rule for the generalised axiom $\bot$, as it is never satisfied.

Validating Axiom Descriptions $\boxed{s, B, \gamma \vdash axdesc \Rightarrow \{\}}$

$$\frac{s, B, \gamma_1 \vdash specexp \Rightarrow \{\} \qquad \langle s, B, \gamma_2 \vdash axdesc \Rightarrow \{\}\rangle}{s, B, \gamma_1\langle\cdot\gamma_2\rangle \vdash specexp\ \langle \texttt{and}\ axdesc\rangle \Rightarrow \{\}} \tag{291}$$

Validating Specification Expressions $\boxed{s, B, \gamma \vdash specexp \Rightarrow \{\}}$

$$\frac{s, B, \gamma_1 \vdash strdec \Rightarrow \mathcal{E} \qquad \dfrac{(s', E) \in \mathcal{E}}{s', B + E, \gamma_2 \vdash axexp \Rightarrow \{\}}}{s, B, \gamma_1 \cdot \gamma_2 \vdash \texttt{let}\ strdec\ \texttt{in}\ axexp\ \texttt{end} \Rightarrow \{\}} \tag{292}$$

Axiomatic Expressions $\boxed{s, B, \gamma \vdash axexp \Rightarrow \{\}}$

$$\frac{s, ((E \text{ of } B, (\{\}, \{\})), \gamma) \vdash exp^\bullet \Rightarrow \texttt{true}, s'}{s, B, \gamma \vdash exp^\bullet \Rightarrow \{\}} \tag{293}$$

Comment: An axiomatic expression $exp^\bullet$ holds if it verificates to `true`; hence `false`, non-termination, and exceptions are treated equally here. Verification of an axiomatic expression has no side-effect, i.e. any side-effect that appeared during verification disappears.

Structure Expressions $\boxed{s, B, \gamma \vdash strexp \Rightarrow \mathcal{S}}$

$$\frac{s, B, \gamma \vdash strdec \Rightarrow \mathcal{E}}{s, B, m \cdot \gamma \vdash \texttt{struct}\ strdec\ \texttt{end} \Rightarrow \{(s', (E, m)) \mid (s', E) \in \mathcal{E}\}} \tag{294}$$

$$\frac{B(longstrid) = S}{s, B, \gamma \vdash longstrid \Rightarrow \{(s, S)\}} \tag{295}$$

$$\frac{\begin{array}{c} B(funid) = (strid, B', (N)(IS, \Sigma)) \qquad \varphi' = s^\#(\varphi) \\ \varphi'(IS, \Sigma) = (IS', \Sigma') \qquad s, B, \gamma \vdash strexp \Rightarrow \mathcal{S} \\ \dfrac{(s', S) \in \mathcal{S}}{\exists S', \varphi''.\ s', (N)IS \vdash S \Rightarrow S', \varphi''} \end{array}}{\begin{array}{l} s, B, \varphi \cdot \gamma \vdash funid\ (\ strexp\ ) \ \Rightarrow \\ \quad \{(s'', S') \mid (s', S) \in \mathcal{S}, \\ \quad s', B' \oplus \{strid \mapsto S\}, \Sigma' \vdash strexp^\bullet \Rightarrow \mathcal{R},\ (s'', (\varphi'', S')) \in \mathcal{R}\ \} \end{array}} \tag{296}$$

$$\frac{s, B, \gamma \vdash strdec \Rightarrow \mathcal{E} \qquad \dfrac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{S}.\ s_1, B \oplus E, \gamma' \vdash strexp \Rightarrow \mathcal{S}}}{\begin{array}{l} s, B, \gamma \cdot \gamma' \vdash \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end} \Rightarrow \\ \quad \{(s_2, S) \mid (s_1, E) \in \mathcal{E},\ s_1, B \oplus E, \gamma' \vdash strexp \Rightarrow \mathcal{S},\ (s_2, S) \in \mathcal{S}\} \end{array}} \tag{297}$$

Comments:

(296) It should be emphasised here that *strexp* and $strexp^\bullet$ are different meta-variables; $strexp^\bullet$ is an *arbitrary* structure expression (not containing `?`). There are no explicit restrictions concerning elaboration of $strexp^\bullet$ needed, because they are implicit in the sentence form used, see rule 298.

Structure vs. Signature

$\boxed{s, B, \Sigma \vdash \mathit{strexp} \Rightarrow \mathcal{R}}$

Sentences of this form can be read as: in state s and basis B, the structure expression *strexp* verificates and matches the signature Σ in the ways specified by $\mathcal{R}$. Each member $(s', (S, \varphi))$ of $\mathcal{R}$ consists of the resulting state s', the matching realisation φ, and the structure S obtained from cutting down the verification result of *strexp* to Σ.

$$\frac{\begin{array}{c}\mathrm{Stat}\, B \vdash_{\mathrm{STAT}} \mathit{strexp} \Rightarrow (m, E_{\mathrm{STAT}}), \gamma \\ s_1, B, \gamma \vdash \mathit{strexp} \Rightarrow \mathcal{S}\end{array} \qquad \frac{(s_2, S) \in \mathcal{S}}{\exists S', \varphi.\ s_2, \Sigma \vdash S \Rightarrow S', \varphi}}{\begin{array}{l}s_1, B, \Sigma \vdash \mathit{strexp} \Rightarrow \\ \quad \{(s_2 + \varphi_{\mathrm{Ty}}, (\varphi, S')) \mid (s_2, S) \in \mathcal{S},\ s_2, \Sigma \vdash S \Rightarrow S', \varphi\}\end{array}} \tag{298}$$

Comment: $s_2 + \varphi_{\mathrm{Ty}}$ is the state obtained by extending the type realisation φ_{Ty} of s_2 by the type part of the realisation φ. The effect of using this state instead of s_2 is to make the implementing types known for the purposes of quantification and comparison.

Structure-level Declarations

$\boxed{s, B, \gamma \vdash \mathit{strdec} \Rightarrow \mathcal{E}}$

Notice that the semantic value of a structure declaration is an $\mathcal{E}$, not an $\mathcal{E}/p$.

$$\frac{}{s, (N, F, G, E), \gamma \vdash \mathit{dec} \Rightarrow \{(s', E') \mid s, (E, QI), \gamma \vdash \mathit{dec} \Rightarrow E', s'\}} \tag{299}$$

$$\frac{}{s, B, \gamma \vdash \texttt{axiom}\ \mathit{ax} \Rightarrow \{(s, \{\} \text{ in Env}) \mid \frac{\gamma \succ \gamma' \notin \text{TraceScheme}}{s, B, \gamma' \vdash \mathit{ax} \Rightarrow \{\}}\ \}} \tag{300}$$

$$\frac{s, B, \gamma \vdash \mathit{strbind} \Rightarrow \mathcal{SE}}{s, B, \gamma \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \{(s', SE \text{ in Env}) \mid (s', SE) \in \mathcal{SE}\}} \tag{301}$$

$$\frac{s, B, \gamma_1 \vdash \mathit{strdec}_1 \Rightarrow \mathcal{E} \qquad \frac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{E}'.\ s_1, B \oplus E, \gamma_2 \vdash \mathit{strdec}_2 \Rightarrow \mathcal{E}'}}{\begin{array}{l}s, B, \gamma_1 \cdot \gamma_2 \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow \\ \quad \{(s_2, E') \mid (s_1, E) \in \mathcal{E},\ s_1, B \oplus E, \gamma_2 \vdash \mathit{strdec}_2 \Rightarrow \mathcal{E}',\ (s_2, E') \in \mathcal{E}'\}\end{array}} \tag{302}$$

$$\frac{}{s, B, \gamma \vdash \qquad \Rightarrow \{(s, \{\} \text{ in Env})\}} \tag{303}$$

$$\frac{s, B, \gamma_1 \vdash \mathit{strdec}_1 \Rightarrow \mathcal{E} \qquad \frac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{E}'.\ s_1, B \oplus E, \gamma_2 \vdash \mathit{strdec}_2 \Rightarrow \mathcal{E}'}}{\begin{array}{l}s, B, \gamma_1 \cdot \gamma_2 \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow \\ \{(s_2, E + E') \mid (s_1, E) \in \mathcal{E},\ s_1, B \oplus E, \gamma_2 \vdash \mathit{strdec}_2 \Rightarrow \mathcal{E}',\ (s_2, E') \in \mathcal{E}'\}\end{array}} \tag{304}$$

Comments:

(299) The sentence in the set comprehension is a sentence of the verification semantics for the Core. Notice that every Core declaration (viewed here as a *strdec*) verificates, but it may verificate to an empty set; it may also verificate to a set with more than one element, because QI can be chosen arbitrarily. All Core verifications resulting in packets are ignored.

(300) Verification of an axiom always succeeds: if the axiom "holds", the result is a singleton set containing the empty environment; otherwise it is the empty set. An axiom only holds if it does so for all type instances of its trace.

Axioms

$$\boxed{s, B, \gamma \vdash ax \Rightarrow \{\}}$$

$$\frac{s, B, \gamma \vdash axexp \Rightarrow \{\} \qquad \langle s, B, \gamma' \vdash ax \Rightarrow \{\} \rangle}{s, B, \gamma\langle\cdot\gamma'\rangle \vdash axexp\ \langle \texttt{and}\ ax \rangle \Rightarrow \{\}} \tag{305}$$

Structure Bindings

$$\boxed{s, B, \gamma \vdash strbind \Rightarrow \mathcal{SE}}$$

$$\frac{s, B, \gamma \vdash sglstrbind \Rightarrow \mathcal{SE} \qquad \left\langle \dfrac{(s', SE) \in \mathcal{SE}}{\exists \mathcal{SE}'.\ s', B + \text{names}\, SE, \gamma' \vdash strbind \Rightarrow \mathcal{SE}'} \right\rangle}{\begin{array}{l} s, B, \gamma\langle\cdot\gamma'\rangle \vdash sglstrbind\ \langle \texttt{and}\ strbind \rangle \Rightarrow \\ \quad \{(s'\langle' \rangle, SE\langle + SE' \rangle) \mid (s', SE) \in \mathcal{SE}\ \langle\, , \\ \qquad s', B + \text{names}\, SE, \gamma' \vdash strbind \Rightarrow \mathcal{SE}',\ (s'', SE') \in \mathcal{SE}' \rangle\ \} \end{array}} \tag{306}$$

Single Structure Bindings

$$\boxed{s, B, \gamma \vdash sglstrbind \Rightarrow \mathcal{SE}}$$

$$\frac{s, \text{Inter}\, B, \gamma \vdash psigexp \Rightarrow \Sigma \qquad s, B, \Sigma \vdash strexp \Rightarrow \mathcal{R}}{\begin{array}{l} s, B, \gamma \cdot \gamma' \vdash strid\ \texttt{:}\ psigexp\ \texttt{=}\ strexp \Rightarrow \\ \quad \{(s', \{strid \mapsto S'\}) \mid s, B, \Sigma \vdash strexp^\bullet \Rightarrow \mathcal{R}',\ (s', (\varphi, S')) \in \mathcal{R}'\ \} \end{array}} \tag{307}$$

$$\frac{s, \text{Inter}\, B, \gamma \vdash psigexp \Rightarrow \Sigma}{\begin{array}{l} s, B, \gamma \vdash strid\ \texttt{:}\ psigexp\ \texttt{=}\ \texttt{?} \Rightarrow \\ \quad \{(s', \{strid \mapsto S\}) \mid s, B, \Sigma \vdash strexp^\bullet \Rightarrow \mathcal{R},\ (s', (\varphi, S)) \in \mathcal{R}\ \} \end{array}} \tag{308}$$

$$\frac{s, B, \gamma \vdash strexp \Rightarrow \mathcal{S}}{s, B, \gamma \vdash strid\ \texttt{=}\ strexp \Rightarrow \{(s', \{strid \mapsto S\}) \mid (s', S) \in \mathcal{S}\}} \tag{309}$$

Comments:

(307) The second premise implicitly requires *every* interpretation of *strexp* to match the signature Σ (see rule 298). The resulting set $\mathcal{R}$ is then ignored and the result is the union of all expressible sets of structures (since $strexp^\bullet$ is arbitrary) matching the signature.

Notice that the trace γ' is thrown away, although the verification of *strexp* re-builds it later using the judgements of the static semantics. The reason for this arrangement is that judgements of the form $s, B, \Sigma \vdash \mathit{strexp} \Rightarrow \mathcal{R}$ are also used for cases in which there is no trace readily available, e.g. rules 308 and 296.

Signature Expressions

$\boxed{s, IB, \gamma \vdash \mathit{sigexp} \Rightarrow IS}$

$$\frac{s, IB, \gamma \vdash \mathit{spec} \Rightarrow I}{s, IB, m \cdot \gamma \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (I, m)} \tag{310}$$

$$\frac{IB(\mathit{sigid}) \geq_{\varphi} IS}{s, IB, \varphi \vdash \mathit{sigid} \Rightarrow IS} \tag{311}$$

Principal Signatures

$\boxed{s, IB, \gamma \vdash \mathit{psigexp} \Rightarrow \Sigma}$

$$\frac{N \cap N \text{ of } IB = \emptyset \qquad s, IB, \gamma \vdash \mathit{sigexp} \Rightarrow IS}{s, IB, (N)\gamma \vdash \mathit{sigexp} \Rightarrow (N)IS} \tag{312}$$

Comments:

(312) The rôle of this rule is similar to that of rule 65. Principality of $(N)IS$ does not have to be imposed because it is implicitly satisfied, as the derivation of $s, IB, \gamma \vdash \mathit{sigexp} \Rightarrow IS$ uses the same realisations as the corresponding derivation of $B_{\text{STAT}} \vdash_{\text{STAT}} \mathit{sigexp} \Rightarrow S_{\text{STAT}}, \gamma$ when determining the principal signature for rule 65.

Signature Declarations

$\boxed{s, IB, \gamma \vdash \mathit{sigdec} \Rightarrow G}$

$$\frac{s, IB, \gamma \vdash \mathit{sigbind} \Rightarrow G}{s, IB, \gamma \vdash \texttt{signature}\ \mathit{sigbind} \Rightarrow G} \tag{313}$$

$$\frac{}{s, IB, \gamma \vdash \qquad \Rightarrow \{\}} \tag{314}$$

$$\frac{s, IB, \gamma_1 \vdash \mathit{sigdec}_1 \Rightarrow G_1 \qquad s, IB + G_1, \gamma_2 \vdash \mathit{sigdec}_2 \Rightarrow G_2}{s, IB, \gamma_1 \cdot \gamma_2 \vdash \mathit{sigdec}_1\ \langle ; \rangle\ \mathit{sigdec}_2 \Rightarrow G_1 + G_2} \tag{315}$$

Signature Bindings

$\boxed{s, IB, \gamma \vdash \mathit{sigbind} \Rightarrow G}$

$$\frac{s, IB, \gamma \vdash \mathit{psigexp} \Rightarrow \Sigma \qquad \langle s, IB, \gamma' \vdash \mathit{sigbind} \Rightarrow G \rangle}{s, IB, \gamma\langle \cdot \gamma' \rangle \vdash \mathit{sigid}\ \texttt{=}\ \mathit{psigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \Rightarrow \{\mathit{sigid} \mapsto \Sigma\}\ \langle + G \rangle} \tag{316}$$

Specifications

$\boxed{s, IB, \gamma \vdash spec \Rightarrow I}$

$$\frac{}{s, IB, VE_{\text{STAT}} \vdash \texttt{val}\ valdesc \Rightarrow s^{\#}(VE_{\text{STAT}})\ \text{in Int}} \tag{317}$$

$$\frac{}{s, IB, TE \vdash \texttt{type}\ typdesc \Rightarrow s^{\#}(TE)\ \text{in Int}} \tag{318}$$

$$\frac{}{s, IB, TE \vdash \texttt{eqtype}\ typdesc \Rightarrow s^{\#}(TE)\ \text{in Int}} \tag{319}$$

$$\frac{}{\begin{array}{c} s, IB, (TE, VE_{\text{STAT}})\ \text{in Env}_{\text{STAT}} \vdash \texttt{datatype}\ datdesc \Rightarrow \\ (s^{\#}(TE), s^{\#}(VE_{\text{STAT}}))\ \text{in Int} \end{array}} \tag{320}$$

$$\frac{}{\begin{array}{c} s, IB, VE_{\text{STAT}}\ \text{in Env}_{\text{STAT}} \vdash \texttt{exception}\ exdesc \Rightarrow \\ (s^{\#}(VE_{\text{STAT}}))\ \text{in Int} \end{array}} \tag{321}$$

$$\frac{}{s, IB, \gamma \vdash \texttt{axiom}\ axdesc \Rightarrow (B\ \text{of}\ IB, \gamma, axdesc)\ \text{in Int}} \tag{322}$$

$$\frac{s, IB, \gamma \vdash strdesc \Rightarrow IE}{s, IB, \gamma \vdash \texttt{structure}\ strdesc \Rightarrow IE\ \text{in Int}} \tag{323}$$

$$\frac{}{s, IB, \gamma \vdash \texttt{sharing}\ shareq \Rightarrow \{\}\ \text{in Int}} \tag{324}$$

$$\frac{\begin{array}{c} IB = (N, F, G, IE, E) \qquad s, IB, \gamma_1 \vdash spec_1 \Rightarrow I_1 \\ IB' = (N \cup \text{names}\, I_1, F, G, IE + IE\ \text{of}\ I_1, E \setminus I_1) \qquad s, IB', \gamma_2 \vdash spec_2 \Rightarrow I_2 \\ I_2 = (IE, TE, VE_{\text{STAT}}, A) \qquad N_1 = \text{names}\, I_2 \setminus \text{names}\, I_1 \setminus N \\ I_3 = (IE, TE, VE_{\text{STAT}}, (N_1, B\ \text{of}\ IB)(I_1, A)) \end{array}}{s, IB, \gamma_1 \cdot \gamma_2 \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow I_3} \tag{325}$$

$$\frac{IB(longstrid_1) = (I_1, m_1) \cdots IB(longstrid_n) = (I_n, m_n)}{s, IB, \gamma \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow I_1 \oplus_{IB} \cdots \oplus_{IB} I_n} \tag{326}$$

$$\frac{IB(sigid_1) \geq_{\varphi_1} (I_1, m_1) \cdots IB(sigid_n) \geq_{\varphi_n} (I_n, m_n)}{s, IB, \varphi_1 \cdot \ldots \cdot \varphi_n \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow I_1 \oplus_{IB} \cdots \oplus_{IB} I_n} \tag{327}$$

$$\frac{}{s, IB, \gamma \vdash \qquad \Rightarrow \{\}\ \text{in Int}} \tag{328}$$

$$\frac{\begin{array}{c} IB = (N, F, G, IE, E) \qquad s, IB, \gamma_1 \vdash spec_1 \Rightarrow I_1 \\ IB' = (N \cup \text{names}\, I_1, F, G, IE + IE\ \text{of}\ I_1, E \setminus I_1) \qquad s, IB', \gamma_2 \vdash spec_2 \Rightarrow I_2 \end{array}}{s, IB, \gamma_1 \cdot \gamma_2 \vdash spec_1\ \langle ; \rangle\ spec_2 \Rightarrow I_1 \oplus_{IB} I_2} \tag{329}$$

Comments:

(317)–(321) The various traces in these rules have to be interpreted in the current state to make choices for question mark types known for verification of phrases that depend on these. Notice also that we use $s^{\#}$ and not $s^{\#\#}$, i.e. specifications in EML do not see the implementing types from other structure bindings.

(324) All the necessary sharing has already been checked in the static analysis, therefore this rule needs no premise.

(325),(329) Notice that the E component of IB shrinks while the IE component increases. E contains global objects which can be hidden by specifications (hence the shrinking), while IE contains interface of global structures as well as of specified structures.

Structure Descriptions

$\boxed{s, IB, \gamma \vdash \mathit{strdesc} \Rightarrow IE}$

$$\frac{s, IB, \gamma \vdash \mathit{sigexp} \Rightarrow IS \qquad \langle s, IB, \gamma' \vdash \mathit{strdesc} \Rightarrow IE \rangle}{s, IB, \gamma\langle\cdot\gamma'\rangle \vdash \mathit{strid} : \ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{strdesc}\rangle \Rightarrow \{\mathit{strid} \mapsto IS\}\ \langle + \ IE\rangle} \tag{330}$$

Functor Declarations

$\boxed{s, B, \gamma \vdash \mathit{fundec} \Rightarrow F}$

$$\frac{s, B, \gamma \vdash \mathit{funbind} \Rightarrow F}{s, B, \gamma \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow F} \tag{331}$$

$$\frac{}{s, B, \gamma \vdash \qquad \Rightarrow \{\}} \tag{332}$$

$$\frac{s, B, \gamma_1 \vdash \mathit{fundec}_1 \Rightarrow F_1 \qquad s, B + F_1, \gamma_2 \vdash \mathit{fundec}_2 \Rightarrow F_2}{s, B, \gamma_1 \cdot \gamma_2 \vdash \mathit{fundec}_1\ \langle ; \rangle\ \mathit{fundec}_2 \Rightarrow F_1 + F_2} \tag{333}$$

Functor Bindings

$\boxed{s, B, \gamma \vdash \mathit{funbind} \Rightarrow F}$

$$\frac{\begin{array}{c} IB = \mathrm{Inter}\, B \qquad \gamma = \gamma_1 \cdot \gamma_2 \cdot \gamma_3\langle\cdot\gamma_4\rangle \\ s, IB, \gamma_1 \vdash \mathit{psigexp} \Rightarrow \Sigma \qquad \Sigma = (N)IS \qquad N \cap N \text{ of } IB = \emptyset \\ s, IB \oplus \{\mathit{strid} \mapsto IS\}, \gamma_2 \vdash \mathit{psigexp}' \Rightarrow \Sigma' \\ \langle s, B, \gamma_4 \vdash \mathit{funbind} \Rightarrow F\rangle \qquad \dfrac{s, B, \Sigma \vdash \mathit{strexp}^{\bullet} \Rightarrow \mathcal{R} \qquad (s', (\varphi, S)) \in \mathcal{R}}{\exists \mathcal{R}'.\ s', B \oplus \{\mathit{strid} \mapsto S\}, \Sigma' \vdash \mathit{strexp} \Rightarrow \mathcal{R}'} \end{array}}{\begin{array}{c} s, B, \gamma \vdash \mathit{funid}\ (\ \mathit{strid} : \mathit{psigexp}\ )\ [:\ \mathit{psigexp}'\ ]\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (\mathit{strid}, B, (N)(IS, \Sigma'))\}\ \langle + \ F\rangle \end{array}} \tag{334}$$

$$\frac{\begin{array}{c} IB = \mathrm{Inter}\, B \qquad \gamma = \gamma_1 \cdot \gamma_2\langle\cdot\gamma_3\rangle \\ s, IB, \gamma_1 \vdash \mathit{psigexp} \Rightarrow \Sigma \qquad \Sigma = (N)IS \qquad N \cap N \text{ of } IB = \emptyset \\ s, IB \oplus \{\mathit{strid} \mapsto IS\}, \gamma_2 \vdash \mathit{psigexp}' \Rightarrow \Sigma' \qquad \langle s, B, \gamma_3 \vdash \mathit{funbind} \Rightarrow F\rangle \end{array}}{\begin{array}{c} s, B, \gamma \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{psigexp}\ \texttt{)}\ \texttt{:}\ \mathit{psigexp}'\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (\mathit{strid}, B, (N)(IS, \Sigma'))\}\ \langle +F\rangle \end{array}} \tag{335}$$

Comments:

(334) The rule in the premise ensures that the argument in a functor application can safely be replaced by anything satisfying the input signature (including generalised axioms) of the functor. "Safely" means here that the functor application is guaranteed to verificate.

(334),(335) There is no difference in the result of these two rules: rule 334 simply has an additional check that *strexp* satisfies the signature $\mathit{psigexp}'$ for any valid functor argument.

Top-level Declarations

$\boxed{s, B, \gamma \vdash \mathit{topdec} \Rightarrow \mathcal{B}}$

$$\frac{s, B, \gamma \vdash \mathit{strdec} \Rightarrow \mathcal{E}}{s, B, \gamma \vdash \mathit{strdec} \Rightarrow\ \{(s', (\mathrm{names}\, E, E)\ \mathrm{in\ Basis}) \mid (s', E) \in \mathcal{E}\}} \tag{336}$$

$$\frac{s, \mathrm{Inter}\, B, \gamma \vdash \mathit{sigdec} \Rightarrow G}{s, B, \gamma \vdash \mathit{sigdec} \Rightarrow\ \{(s, (\mathrm{names}\, G, G)\ \mathrm{in\ Basis})\}} \tag{337}$$

$$\frac{s, B, \gamma \vdash \mathit{fundec} \Rightarrow F}{s, B, \gamma \vdash \mathit{fundec} \Rightarrow\ \{(s, (\mathrm{names}\, F, F)\ \mathrm{in\ Basis})\}} \tag{338}$$

10 Programs

The phrase classes FullProgram and Program of programs are defined as follows:

$$\begin{array}{rcl} \mathit{program} & ::= & \mathit{topdec}\ ;\ \langle \mathit{program} \rangle \\ \mathit{fullprogram} & ::= & \mathit{program} \end{array}$$

The variable *topdec* above refers to top-level declarations of the Full language. As the semantic rules shown so far only operate on the Bare language, we have to translate such a *topdec* into a top-level declaration of the Bare language — the rules for this purpose are 478–481 in Appendix B.

Semantic objects in this section are the semantic objects for the verification semantics; objects coming from other parts of the semantics are accordingly indexed, for example C_{DER} is a context of the semantics for derived forms. Similarly, we attribute each turnstile with the part of the semantics that it refers to — a missing index means "program semantics".

Consider a sentence of the form $\vdash \mathit{fullprogram} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}$. The following main situations can arise:

- There is no $\mathcal{B}, C_{\mathrm{DER}}$ such that $\vdash \mathit{fullprogram} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}$ holds. This is the case if *fullprogram* either contains static errors (static semantics, derived forms) or an interface error of the verification semantics for modules, i.e. a structure (resp. functor) in *fullprogram* does not match its signature (resp. signatures).

- We have $\vdash \mathit{fullprogram} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}$ but $\mathcal{B} = \emptyset$. This situation arises if *fullprogram* contains a the structure or functor declaration which is inconsistent, i.e. which has an empty class of models. This inconsistency can arise in a number of ways, most typically if a stated axiom does not hold in any model or if a value declaration does not terminate or raises an exception.

- Otherwise all of the structure and functor declarations in *fullprogram* are consistent and each $(s, B) \in \mathcal{B}$ (together with C_{DER}) can be seen as a model of *fullprogram*.

Top-level Declarations

$$\boxed{s, B, C_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \mathcal{B}, C'_{\mathrm{DER}}}$$

$$\frac{\neg\exists C'_{\mathrm{DER}}, \mathit{topdec}'.\ C_{\mathrm{DER}} \vdash_{\mathrm{DER}} \mathit{topdec} \Rightarrow \mathit{topdec}', C'_{\mathrm{DER}}}{s, B, C_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \{(s, B)\},\ C_{\mathrm{DER}}} \tag{339}$$

$$\frac{\begin{array}{c} C_{\mathrm{DER}} \vdash_{\mathrm{DER}} \mathit{topdec} \Rightarrow \mathit{topdec}', C'_{\mathrm{DER}} \\ \neg\exists B_{\mathrm{STAT}}, \gamma.\ \mathrm{Stat}\, B \vdash_{\mathrm{STAT}} \mathit{topdec}' \Rightarrow B_{\mathrm{STAT}}, \gamma \end{array}}{s, B, C_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \{(s, B)\},\ C_{\mathrm{DER}}} \tag{340}$$

$$\frac{C_{\mathrm{DER}} \vdash_{\mathrm{DER}} \mathit{topdec} \Rightarrow \mathit{topdec}', C'_{\mathrm{DER}} \qquad \dfrac{\mathrm{Stat}\, B \vdash_{\mathrm{STAT}} \mathit{topdec}' \Rightarrow B'_{\mathrm{STAT}}, \gamma'}{(\mathrm{names}\,\gamma \setminus N \,\mathrm{of}\, B)\gamma \succ \gamma'} \qquad \mathrm{Stat}\, B \vdash_{\mathrm{STAT}} \mathit{topdec}' \Rightarrow B_{\mathrm{STAT}}, \gamma \qquad s, B, \gamma \vdash_{\mathrm{VER}} \mathit{topdec}' \Rightarrow \mathcal{B}}{s, B, C_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \{(s', B \oplus B') \mid (s', B') \in \mathcal{B}\},\ C_{\mathrm{DER}} + C'_{\mathrm{DER}}} \tag{341}$$

Comments:

(340) A failing elaboration has no effect whatever. Even derived form declarations are "undone". In contrast to SML there is no rule for dealing with raised exceptions, because packets cannot escape core level verification in EML.

(341) The rule in the premise ensures that the top-level declaration is verificated with a principal trace.

Programs

$$\boxed{s, B, C_{\mathrm{DER}} \vdash \mathit{program} \Rightarrow \mathcal{B}, C'_{\mathrm{DER}}}$$

$$\frac{s, B, C_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \mathcal{B}, C'_{\mathrm{DER}} \qquad \left\langle \dfrac{(s', B') \in \mathcal{B}}{\exists \mathcal{B}'.\ s', B', C'_{\mathrm{DER}} \vdash \mathit{program} \Rightarrow \mathcal{B}', C''_{\mathrm{DER}}} \right\rangle}{\begin{array}{c} s, B, C_{\mathrm{DER}} \vdash \mathit{topdec}\ ;\ \langle \mathit{program} \rangle \Rightarrow \{(s'\langle' \rangle, B'\langle' \rangle) \mid (s', B') \in \mathcal{B}, \\ \langle s', B', C'_{\mathrm{DER}} \vdash \mathit{program} \Rightarrow \mathcal{B}', C''_{\mathrm{DER}}, (s'', B'') \in \mathcal{B}' \rangle\},\ C'\langle' \rangle_{\mathrm{DER}} \end{array}} \tag{342}$$

Comment: Notice that C''_{DER} is scoped at the main rule, see Section 1.2 — the resulting context for derived forms is not model-dependent and is the same for all $(s', B') \in \mathcal{B}$. Another consequence of this scoping is that C''_{DER} is arbitrary if $\mathcal{B} = \emptyset$.

Full Program

$$\boxed{\vdash \mathit{fullprogram} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}}$$

$$\frac{s_0, B_0, C^0_{\mathrm{DER}} \vdash \mathit{program} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}}{\vdash \mathit{program} \Rightarrow \mathcal{B}, C_{\mathrm{DER}}} \tag{343}$$

Comment: This rule connects the program semantics with the initial state s_0, the initial verification basis B_0, and the inital context for derived forms C^0_{DER}.

A Appendix: Full Grammar

This section gives the full grammar of Extended ML, which includes the syntax for the Core, the syntax for Modules and the derived forms. Syntactic phrases of the Full Language contain identifiers *without* status labels, i.e. the class Id takes the rôle of the classes Var, StrId, etc.

The Full language uses the same names for its phrase classes as the Bare language, see figures 3 and 7 on pages 11 and 17, respectively. Concerning the syntax of expressions, two additional subclasses of the phrase class Exp are introduced, namely AppExp (application expressions) and InfExp (infix expressions). The inclusion relation among the four classes is as follows:

$$\text{AtExp} \subset \text{AppExp} \subset \text{InfExp} \subset \text{Exp}$$

The effect is that certain phrases, such as "`2 + if` $\cdots$ `then` $\cdots$ `else` $\cdots$ ", are now disallowed. An analogous construction applies to patterns, i.e. we also have additional phrase classes AppPat and InfPat, etc. Concerning identifiers, there are the phrase classes Id (and LongId, analogous to Section 2.4), Lab and TyVar. Each $id \in$ Id is either alphanumeric (but not starting with a prime) or symbolic. Members of Id are considered to be lexical items. Another additional phrase class is D (digits): each $d \in D$ is a character between 0 and 9 and is also regarded as a lexical item.

The grammatical conventions are similar to Section 2, namely:

- The brackets $\langle\ \rangle$ enclose optional phrases.
- For any syntax class X (over which x ranges) we define the syntax class Xseq (over which *xseq* ranges) as follows:

xseq	::=	x	(singleton sequence)
			(empty sequence)
		$(x_1, \cdots, x_n)$	(sequence, $n \geq 1$)

  Note that the "$\cdots$" used here, a meta-symbol indicating syntactic repetition, must not be confused with "`...`" which is a reserved word of the language. To range over all three alternatives for sequences in semantic rules we write $x_1 \cdots x_n$ (with $n \geq 0$), which suppresses the syntactic commas and parentheses. The ambiguity for $n = 1$ will be harmless whenever we use this notation.

- Alternative forms for each phrase class are in order of decreasing precedence. This precedence resolves ambiguity in parsing in the following way. Suppose that a phrase class *phrase* has several alternative forms $F_1 \cdots F_n$. If a lexical sequence $L_1 \cdots L_k$ reduces to more than one of the F_i, then it reduces to *phrase* via the F_i with lowest precedence. Example: The parsing of the sequence

  `if` exp_1 `then` exp_2 `else` exp_3 `handle` *match*

is determined by the above principle. Because `if`-expressions have lower precedence than `handle`-expressions, the sequence parses as

$$\texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ (exp_3\ \texttt{handle}\ match)$$

Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases. In particular, the purpose is not to prevent a phrase, which is an instance of a form with higher precedence, having a constituent which is an instance of a form with lower precedence. Thus for example

$$\texttt{if}\ \cdots\ \texttt{then case}\ \cdots\ \texttt{of}\ \cdots\ \texttt{else case}\ \cdots\ \texttt{of}\ \cdots$$

is quite admissible, and will be parsed as

$$\texttt{if}\ \cdots\ \texttt{then}\ (\texttt{case}\ \cdots\ \texttt{of}\ \cdots)\ \texttt{else}\ (\texttt{case}\ \cdots\ \texttt{of}\ \cdots)$$

- Longest match: Suppose F_1F_2 is an alternative form of a phrase class. A natural number i is called a *split index* w.r.t. F_1F_2 for a lexical sequence $L_1 \cdots L_k$ if $0 \leq i \leq k$ and $L_1 \cdots L_i$ reduces to F_1 and $L_{i+1} \cdots L_k$ reduces to F_2. If for a given lexical sequence $L = L_1 \cdots L_k$ there are different split indices w.r.t. F_1F_2, then L reduces to F_1F_2 by reducing $L_1 \cdots L_j$ to F_1, where j is either the maximal split index, or — iff the alternative form is labelled (R), indicating a right associative infix construct — the minimal split index.

- For any syntax class X (over which x ranges) we define the syntax class $\text{X}^\bullet$ (over which $x^\bullet$ ranges) as the same as X, except that phrases of class $\text{X}^\bullet$ may not contain `?`.

atexp	::=	*scon*	special constant
		⟨`op`⟩ *longid*	long identifier
		`{` ⟨*exprow*⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		(exp_1 `,` ⋯ `,` exp_n)	n-tuple, $n \geq 2$
		[exp_1 `,` ⋯ `,` exp_n]	list, $n \geq 0$
		(exp_1 `;` ⋯ `;` exp_n)	sequence, $n \geq 2$
		`let` *dec* `in` exp_1 `;` ⋯ `;` exp_n `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	
		`?`	undefined value
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
appexp	::=	*atexp*	
		appexp atexp	application expression
infexp	::=	*appexp*	
		$infexp_1$ *id* $infexp_2$	infix expression
exp	::=	*infexp*	
		exp `:` *ty*	typed
		$exp_1^\bullet$ `==` $exp_2^\bullet$	comparison (R)
		$exp_1^\bullet$ `=/=` $exp_2^\bullet$	comparison (R)
		`exists` $match^\bullet$	existential quantifier
		`forall` $match^\bullet$	universal quantifier
		$exp^\bullet$ `terminates`	convergence predicate
		$exp^\bullet$ `proper`	definedness predicate
		exp_1 `andalso` exp_2	conjunction
		exp_1 `orelse` exp_2	disjunction
		exp_1 `implies` exp_2	implication (R)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		exp `raises` *match*	test for exception
		exp `raises` *pat*	test for exception
		`if` exp_1 `then` exp_2 `else` exp_3	conditional
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 19: Grammar: Expressions and Matches

dec	::=	`val` *valbind*	value declaration
		`fun` *fvalbind*	function declaration
		`type` *typbind*	type declaration
		`eqtype` *typbind*	equality type declaration
		`datatype` *datbind*	datatype declaration
		⟨`withtype` *typbind*$^\bullet$⟩	
		`abstype` *datbind*	abstype declaration
		⟨`withtype` *typbind*$^\bullet$⟩	
		`with` *dec* `end`	
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longid_1 \cdots longid_n$	open declaration, $n \geq 1$
		dec_1 ⟨;⟩ dec_2	sequential declaration
			empty declaration
		`infix` ⟨d⟩ $id_1 \cdots id_n$	infix directive, $n \geq 1$
		`infixr` ⟨d⟩ $id_1 \cdots id_n$	infix directive, $n \geq 1$
		`nonfix` $id_1 \cdots id_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
fvalbind	::=	*funcbind* ⟨`and` *fvalbind*⟩	function declarations
funcbind	::=	*fpat* ⟨: *ty*⟩ `=` *exp* ⟨`|` *funcbind*⟩	single function
typbind	::=	*tyvarseq id* `=` *ty* ⟨`and` *typbind*⟩	type binding
		tyvarseq id `=` `?` ⟨`and` *typbind*⟩	question mark type binding
datbind	::=	*tyvarseq id* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	*id* ⟨`of` *ty*⟩ ⟨`|` *conbind*⟩	
exbind	::=	*id* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		id `=` *longid* ⟨`and` *exbind*⟩	

Figure 20: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*longid*	
		`{` ⟨*patrow*⟩ `}`	record
		`()`	0-tuple
		$($*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$$)$	n-tuple, $n \geq 2$
		$[$*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$$]$	list, $n \geq 0$
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
		id⟨`:` *ty*⟩ ⟨`as` *pat*⟩ ⟨`,` *patrow*⟩	label as variable
apppat	::=	⟨`op`⟩*longid atpat*	construction
		atpat	atomic
infpat	::=	*apppat*	application or atomic
		infpat$_1$ *id infpat*$_2$	infix construction
pat	::=	*infpat*	
		pat `:` *ty*	typed
		⟨`op`⟩*id* ⟨`:` *ty*⟩ `as` *pat*	layered
fpat	::=	⟨`op`⟩*id atpat*$_1 \cdots$*atpat*$_n$	*fvalbind* pattern sequence
		$($*atpat*$_1$ *id atpat*$'_1$$)$ *atpat*$_2 \cdots$*atpat*$_n$	
		atpat$_1$ *id atpat*$'_1$	

Figure 21: Grammar: Patterns

ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type
		tyseq longid	type construction
		ty$_1$ `*` $\cdots$ `*` *ty*$_n$	tuple type, $n \geq 2$
		ty `->` *ty*$'$	function type (R)
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 22: Grammar: Type expressions

strexp	::=	`struct` *strdec* `end`	generative
		longid	structure identifier
		id `(` *strexp* `)`	functor application
		id `(` *strdec* `)`	
		`let` *strdec* `in` *strexp* `end`	local declaration
strdec	::=	*dec*	declaration
		`axiom` *ax* ⟨`withtype` *typbind*$^\bullet$⟩	axiom
		`structure` *strbind*	structure
		`local` *strdec*$_1$ `in` *strdec*$_2$ `end`	local
		strdec$_1$ ⟨`;`⟩ *strdec*$_2$	sequential
			empty
ax	::=	*axexp* ⟨`and` *ax*⟩	axioms
axexp	::=	*exp*$^\bullet$	axiomatic expression
strbind	::=	*sglstrbind* ⟨`and` *strbind*⟩	structure binding
sglstrbind	::=	*id* `:` *psigexp* `=` *strexp*	single structure binding
		id `:` *psigexp* `=` `?`	undefined structure binding
		id `=` *strexp*	unguarded structure binding
sigexp	::=	`sig` *spec* `end`	generative
		id	signature identifier
psigexp	::=	*sigexp*	principal signature
sigdec	::=	`signature` *sigbind*	single
		sigdec$_1$ ⟨`;`⟩ *sigdec*$_2$	sequential
			empty
sigbind	::=	*id* `=` *psigexp* ⟨`and` *sigbind*⟩	

Figure 23: Grammar: Structure and Signature Expressions

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc*	datatype
		`exception` *exdesc*	exception
		`axiom` *axdesc*	axiom
		`structure` *strdesc*	structure
		`sharing` *shareq*	sharing
		`local` $spec_1$ `in` $spec_2$ `end`	local
		`open` $longid_1 \cdots longid_n$	open ($n \geq 1$)
		`include` $id_1 \cdots id_n$	include ($n \geq 1$)
		$spec_1$ $\langle$`;`$\rangle$ $spec_2$	sequential
			empty
valdesc	::=	*id* : *ty* $\langle$`and` *valdesc*$\rangle$	
typdesc	::=	*tyvarseq id* $\langle$`and` *typdesc*$\rangle$	
datdesc	::=	*tyvarseq id* `=` *condesc* $\langle$`and` *datdesc*$\rangle$	
condesc	::=	*id* $\langle$`of` *ty*$\rangle$ $\langle$`|` *condesc*$\rangle$	
exdesc	::=	*id* $\langle$`of` *ty*$\rangle$ $\langle$`and` *exdesc*$\rangle$	
axdesc	::=	*specexp* $\langle$`and` *axdesc*$\rangle$	
specexp	::=	`let` *strdec* `in` *axexp* `end`	
		$exp^\bullet$	
strdesc	::=	*id* : *sigexp* $\langle$`and` *strdesc*$\rangle$	
shareq	::=	$longid_1$ `=` $\cdots$ `=` $longid_n$	structure sharing ($n \geq 2$)
		`type` $longid_1$ `=` $\cdots$ `=` $longid_n$	type sharing ($n \geq 2$)
		$shareq_1$ `and` $shareq_2$	multiple

Figure 24: Grammar: Specifications

fundec	::=	`functor` *funbind*	single
		$fundec_1$ ⟨`;`⟩ $fundec_2$	sequence
			empty
funbind	::=	*id* `(` *id′* `:` *psigexp* `):` *psigexp′* `=` *strexp* ⟨`and` *funbind*⟩	functor binding
		id `(` *spec* `):` *psigexp* `=` *strexp* ⟨`and` *funbind*⟩	
		id `(` *id′* `:` *psigexp* `):` *psigexp′* `= ?` ⟨`and` *funbind*⟩	undefined functor binding
		id `(` *spec* `):` *psigexp* `= ?` ⟨`and` *funbind*⟩	
topdec	::=	*strdec*	structure-level declaration
		sigdec	signature declaration
		fundec	functor declaration
		exp	expression at top-level
program	::=	*topdec* `;` ⟨*program*⟩	
fullprogram	::=	*program*	

Note: No *topdec* may contain, as an initial segment, a shorter top-level declaration followed by a semicolon.

Figure 25: Grammar: Functors and Top-level Declarations

A.1 Syntactic Restrictions

The syntactic restrictions mentioned in sections 2.9 and 3.5 apply analogously to the full grammar.

Further restrictions are expressed in the next section, the semantics of derived forms.

B Appendix: Derived Forms

The rules in this section translate syntactical phrases of the full grammar into syntactical phrases of the Bare language.

B.1 Semantic Objects

The semantic objects of this section are the syntactic phrases of Bare and Full language and the objects in Figure 26.

$$
\begin{array}{rcl}
st & \in & \text{Status} = \{\mathtt{v}, \mathtt{c}, \mathtt{e}, \mathtt{g}, \mathtt{s}, \mathtt{f}, \mathtt{t}\} \\
fix & \in & \text{Fixity} = \{\mathtt{l}, \mathtt{r}, \mathtt{n}\} \\
n & \in & \mathcal{N} = \{0, 1, 2, \ldots\} \\
VE & \in & \text{VarEnv} = \text{Id} \xrightarrow{\text{fin}} (\mathcal{N} \times \text{Fixity}) \\
StE & \in & \text{StatEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Status} \\
TE & \in & \text{TyEnv} = \text{Id} \xrightarrow{\text{fin}} (\text{TypeFcn} \uplus \{\text{FAIL}\}) \\
\Lambda\alpha^{(k)}.ty & \in & \text{TypeFcn} = \uplus_{k \geq 0} \text{TyVar}^k \times \text{Ty} \\
SE & \in & \text{StrEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Env} \\
F & \in & \text{FunEnv} = \text{Id} \xrightarrow{\text{fin}} (\text{Env} \times \text{Env}) \\
G & \in & \text{SigEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Env} \\
(SE, TE, StE) \text{ or } E & \in & \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{StatEnv} \\
(E, VE, F, G) \text{ or } C & \in & \text{Context} = \text{Env} \times \text{VarEnv} \times \text{FunEnv} \times \text{SigEnv} \\
idval & \in & \text{Idval} = \text{Env} \uplus \text{TypeFcn} \uplus \text{Status} \uplus \\
 & & \qquad (\mathcal{N} \times \text{Fixity}) \uplus (\text{Env} \times \text{Env}) \uplus \{\text{FAIL}\} \\
\pi & \in & \text{Context} \to (\text{Id} \xrightarrow{\text{fin}} \text{Idval})
\end{array}
$$

Figure 26: Semantic Objects for Derived Forms

The results of looking up a long identifier in a context (rules 344–346) includes an *identifier value*. The object class IdVal contains the codomains of all environments of a context plus {FAIL}. The identifier value FAIL indicates that the access is unsuccessful, see rule 345. This is also used to generate "fresh" identifiers. Within type environments an identifier is bound to FAIL if this binding does not originate from a `withtype` type binding.

The variable π serves as a placeholder for various projections from a context to its components. A concrete projection is written with an _ indicating the argument position, e.g. (SE of E of _) selects the structure environment of a context.

Given a type expression ty, a sequence of type variables $\alpha_1, \ldots, \alpha_k$ and a sequence of type expressions $ty_1, \ldots, ty_k$, we write $ty\{ty_1/\alpha_1, \ldots, ty_k/\alpha_k\}$ for the result of the simultaneous substitution of the type variables α_i throughout ty by the corresponding type expressions ty_i. The conventions and notations used for

projection, injection and modification in preceding sections are adapted here as well, except that within this section, we write $C \oplus E$ for the context obtained from C by *replacing* its first component by E. To obtain default behaviour for certain forms of environments, we introduce the following notation:

$$\begin{array}{rcll} [StE](id) & = & StE(id) & \text{if } id \in \mathrm{Dom}\, StE \\ & = & \mathtt{v} & \text{otherwise} \\ [VE](id) & = & VE(id) & \text{if } id \in \mathrm{Dom}\, VE \\ & = & (0, \mathtt{n}) & \text{otherwise} \end{array}$$

In other words: $[StE]$ resp. $[VE]$ is an (infinite) map which extends the finite map StE resp. VE by the default value `v` resp. $(0, \mathtt{n})$.

B.2 Identifiers

The Bare language distinguishes different forms of identifiers, e.g. value variables and value constructors, structure identifiers and signature identifiers. Syntactically, identifiers in the classes Var, Con, ExCon, TyCon, StrId, SigId and FunId all belong to the class Id. Within phrases of the Bare language, we distinguish them by their attached identifier status, i.e. id^{st} belongs to the class indicated by st, where `v`, `c`, `e`, `t`, `s`, `g` and `f` refer to Var, Con, ExCon, TyCon, StrId, SigId and FunId, respectively. Thus, any value variable *var* has the form $id^{\mathtt{v}}$ with $id \in$ Id, etc.

Variable environments map identifiers to their fixity (n, fix). This is used for parsing expressions with infix operators: the number n is the precedence of the identifier (when used infix) and *fix* says whether it is left- or right-associative (`l` or `r`, respectively) or non-infix (`n`).

Phrases of the Full language only contain plain identifiers (i.e. without attached status) and the semantic rules in this section which translate the Full into the Bare language also provide this status information.

Given a natural number n, we write $\overline{n}$ for a label *lab* that is a string of digits and that denotes this number when interpreted as a special constant.

B.3 Enrichment

We define a relation $\succ$ on identifier status as $st \succ st' \iff st = st' \vee st' = \mathtt{v}$. We extend this to (status, structure) environments as follows:

$$\begin{array}{rcl} E_1 \succ E_2 & \iff & StE \text{ of } E_1 \succ StE \text{ of } E_2 \wedge SE \text{ of } E_1 \succ SE \text{ of } E_2 \\ SE_1 \succ SE_2 & \iff & \forall id \in \mathrm{Dom}\, SE_2.\ id \in \mathrm{Dom}\, SE_1 \wedge SE_1(id) \succ SE_2(id) \\ StE_1 \succ StE_2 & \iff & \forall id \in \mathrm{Dom}\, StE_2.\ id \in \mathrm{Dom}\, StE_1 \wedge StE_1(id) \succ StE_2(id) \end{array}$$

B.4 Inference Rules

All sentences of this part of the semantics have the form $C, A_1^* \vdash phrase \Rightarrow phrase', A_2^*$, where C is a context, A_1^* and A_2^* are sequences of semantic or syntactic

objects, *phrase* is a phrase of the Full language and *phrase'* is a (corresponding) phrase of the Bare language. Syntactic objects in A_1^* and A_2^* belong to the Bare language. When we have $C, A_1^* \vdash \mathit{phrase} \Rightarrow \mathit{phrase}', A_2^*$, then *phrase'* and A_2^* are determined by C, A_1^*, and *phrase*, modulo choice of fresh identifier names (e.g. as in rule 351).

We do not give all rules explicitly. When we omit the sentence form for a phrase class Phrase, the sentence form is $C \vdash \mathit{phrase} \Rightarrow \mathit{phrase}'$. When we omit rules for an alternative form of a phrase class, then these rules can be determined as follows: if the alternative form in the full grammar is $t_1\ v_1\ t_2 \cdots t_k\ v_k\ t_{k+1}$ ($k \geq 0$), where each t_i is a sequence of lexical items (terminals) and each v_i is a variable ranging over a phrase class (in the Full language), then the missing rule is:

$$\frac{C \vdash v_1 \Rightarrow v_1' \quad \cdots \quad C \vdash v_k \Rightarrow v_k'}{C \vdash t_1\ v_1\ t_2 \cdots t_k\ v_k\ t_{k+1} \Rightarrow t_1\ v_1'\ t_2 \cdots t_k\ v_k'\ t_{k+1}}$$

Long Identifiers

$\boxed{C, \pi, st \vdash \mathit{longid} \Rightarrow \mathit{longid}', \mathit{idval}}$

$$\frac{\mathit{id} \in \mathrm{Dom}(\pi C) \qquad \mathit{idval} = \pi(C)(\mathit{id})}{C, \pi, st \vdash \mathit{id} \Rightarrow \mathit{id}^{st}, \mathit{idval}} \tag{344}$$

$$\frac{\mathit{id} \notin \mathrm{Dom}(\pi C)}{C, \pi, st \vdash \mathit{id} \Rightarrow \mathit{id}^{st}, \mathrm{FAIL}} \tag{345}$$

$$\frac{(\mathit{SE}\text{ of }E\text{ of }C)(\mathit{id}) = E \qquad C \oplus E, \pi, st \vdash \mathit{longid} \Rightarrow \mathit{longid}', \mathit{idval}}{C, \pi, st \vdash \mathit{id}\,.\,\mathit{longid} \Rightarrow \mathit{id}^{\mathtt{S}}\,.\,\mathit{longid}', \mathit{idval}} \tag{346}$$

Comments:

(345) This rule has two purposes. On the one hand, it detects identifiers that are used without being declared. On the other, it is also used to generate fresh identifiers, e.g. in rule 351.

Value Identifiers

$\boxed{C \vdash \langle \mathtt{op} \rangle\ \mathit{longid} \Rightarrow \mathit{longid}', st}$

$$\frac{[\mathit{VE}\text{ of }C](\mathit{id}) = (n, \mathit{fix}) \qquad \mathit{fix} \neq \mathtt{n} \qquad [\mathit{StE}\text{ of }E\text{ of }C](\mathit{id}) = st}{C \vdash \mathtt{op}\ \mathit{id} \Rightarrow \mathit{id}^{st}, st} \tag{347}$$

$$\frac{[\mathit{VE}\text{ of }C](\mathit{id}) = (n, \mathtt{n}) \qquad [\mathit{StE}\text{ of }E\text{ of }C](\mathit{id}) = st}{C \vdash \langle \mathtt{op} \rangle\ \mathit{id} \Rightarrow \mathit{id}^{st}, st} \tag{348}$$

$$\frac{(\mathit{SE}\text{ of }E\text{ of }C)(\mathit{id}) = E \qquad C \oplus E \vdash \mathtt{op}\ \mathit{longid} \Rightarrow \mathit{longid}', st}{C \vdash \langle \mathtt{op} \rangle\ \mathit{id}\,.\,\mathit{longid} \Rightarrow \mathit{id}^{\mathtt{S}}\,.\,\mathit{longid}', st} \tag{349}$$

Comments:

(349) The presence of `op` in the premise, regardless whether it occurs in the conclusion or not, means that identifiers with infix status can be used non-infix in their long form.

Atomic Expressions

$\boxed{C \vdash atexp \Rightarrow atexp'}$

$$\frac{C \vdash \langle \texttt{op} \rangle longid \Rightarrow longid', st}{C \vdash \langle \texttt{op} \rangle longid \Rightarrow longid'} \tag{350}$$

$$\frac{C, StE \text{ of } E \text{ of } _, \texttt{v} \vdash id \Rightarrow var, \text{FAIL}}{C \vdash \texttt{\#}\ lab \Rightarrow\ \texttt{(fn \{}\ lab\texttt{=}var\texttt{,...\}\ =>}\ var\texttt{)}} \tag{351}$$

$$\frac{}{C \vdash\ \texttt{()} \Rightarrow\ \{\}} \tag{352}$$

$$\frac{n \geq 2 \qquad C \vdash exp_1 \Rightarrow exp_1' \quad \cdots \quad C \vdash exp_n \Rightarrow exp_n'}{C \vdash\ \texttt{(}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{)} \Rightarrow\ \texttt{\{1=}exp_1'\texttt{,}\cdots\texttt{,}\overline{n}\texttt{=}exp_n'\texttt{\}}} \tag{353}$$

$$\frac{n \geq 0 \qquad C \vdash \texttt{(}exp_1\texttt{)::}\cdots\texttt{::(}exp_n\texttt{)::nil} \Rightarrow exp}{C \vdash\ \texttt{[}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{]} \Rightarrow\ \texttt{(}exp\texttt{)}} \tag{354}$$

$$\frac{n \geq 1 \qquad C \vdash \texttt{case (}exp_1\texttt{) of _ =>}\ \cdots\ \texttt{case (}exp_n\texttt{) of _ => (}exp\texttt{)} \ \Rightarrow\ exp'}{C \vdash\ \texttt{(}exp_1\texttt{;}\cdots\texttt{;}exp_n\texttt{;}exp\texttt{)} \Rightarrow\ \texttt{(}exp'\texttt{)}} \tag{355}$$

$$\frac{n \geq 1 \qquad C \vdash dec \Rightarrow dec', C' \qquad C + C' \vdash \texttt{(}exp_1\texttt{;}\cdots\texttt{;}exp_n\texttt{)} \Rightarrow atexp}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp_1\texttt{;}\cdots\texttt{;}exp_n\ \texttt{end} \Rightarrow \texttt{let}\ dec'\ \texttt{in}\ atexp\ \texttt{end}} \tag{356}$$

Comments:

(351) The premise selects a fresh identifier *id* and attributes it with status `v`, giving the value variable *var*.

(354) It is assumed here that neither `nil` nor `::` has been rebound, and `::` still has right-associative infix status.

(356) The side-condition $n \geq 1$ means that the rule applies to all `let`-expressions.

Applications

$\boxed{C \vdash appexp \Rightarrow exp}$

$$\frac{C \vdash appexp \Rightarrow exp \qquad C \vdash atexp \Rightarrow atexp'}{C \vdash appexp\ atexp \Rightarrow \texttt{(}exp\texttt{)}\ atexp'} \tag{357}$$

Infix Expressions

$\boxed{C, n, \mathit{fix} \vdash \mathit{infexp} \Rightarrow \mathit{exp}}$

$$\frac{C \vdash \mathit{appexp} \Rightarrow \mathit{exp}}{C, n, \mathit{fix} \vdash \mathit{appexp} \Rightarrow \mathit{exp}} \tag{358}$$

$$\frac{\begin{array}{c}[VE \text{ of } C](id) = (n', \texttt{1}) \qquad n < n' \vee (n = n' \wedge \mathit{fix} = \texttt{1}) \\ [StE \text{ of } E \text{ of } C](id) = st \\ C, n', \texttt{1} \vdash \mathit{infexp}_1 \Rightarrow \mathit{exp}_1 \qquad C, n', \texttt{n} \vdash \mathit{infexp}_2 \Rightarrow \mathit{exp}_2\end{array}}{C, n, \mathit{fix} \vdash \mathit{infexp}_1\ id\ \mathit{infexp}_2 \Rightarrow id^{st}\ \{\texttt{1=}\mathit{exp}_1\texttt{,2=}\mathit{exp}_2\}} \tag{359}$$

$$\frac{\begin{array}{c}[VE \text{ of } C](id) = (n', \texttt{r}) \qquad n < n' \vee (n = n' \wedge \mathit{fix} = \texttt{r}) \\ [StE \text{ of } E \text{ of } C](id) = st \\ C, n', \texttt{n} \vdash \mathit{infexp}_1 \Rightarrow \mathit{exp}_1 \qquad C, n', \texttt{r} \vdash \mathit{infexp}_2 \Rightarrow \mathit{exp}_2\end{array}}{C, n, \mathit{fix} \vdash \mathit{infexp}_1\ id\ \mathit{infexp}_2 \Rightarrow id^{st}\ \{\texttt{1=}\mathit{exp}_1\texttt{,2=}\mathit{exp}_2\}} \tag{360}$$

Comments:

(360) The SML definition makes two *different* right-associative operators of the same precedence associate to the left. This rule makes them associate to the right.

Expressions

$\boxed{C \vdash \mathit{exp} \Rightarrow \mathit{exp}'}$

Not all of the parentheses these rules generate are really necessary; many of them are here only for uniformity of presentation.

$$\frac{C, 0, \mathit{fix} \vdash \mathit{infexp} \Rightarrow \mathit{exp}}{C \vdash \mathit{infexp} \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{361}$$

$$\frac{C \vdash \texttt{not(}\mathit{exp}_1^\bullet\ \texttt{==}\ \mathit{exp}_2^\bullet\texttt{)} \Rightarrow \mathit{exp}}{C \vdash \mathit{exp}_1^\bullet\ \texttt{=/=}\ \mathit{exp}_2^\bullet \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{362}$$

$$\frac{C \vdash \texttt{(}\mathit{exp}^\bullet\ \texttt{terminates) andalso not(}\mathit{exp}^\bullet\ \texttt{raises _)} \Rightarrow \mathit{exp}}{C \vdash \mathit{exp}^\bullet\ \texttt{proper} \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{363}$$

$$\frac{C \vdash \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else false} \Rightarrow \mathit{exp}}{C \vdash \mathit{exp}_1\ \texttt{andalso}\ \mathit{exp}_2 \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{364}$$

$$\frac{C \vdash \texttt{if}\ \mathit{exp}_1\ \texttt{then true else}\ \mathit{exp}_2 \Rightarrow \mathit{exp}}{C \vdash \mathit{exp}_1\ \texttt{orelse}\ \mathit{exp}_2 \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{365}$$

$$\frac{C \vdash \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else true} \Rightarrow \mathit{exp}}{C \vdash \mathit{exp}_1\ \texttt{implies}\ \mathit{exp}_2 \Rightarrow \texttt{(}\mathit{exp}\texttt{)}} \tag{366}$$

$$\frac{C \vdash \texttt{((}\mathit{exp}\ \texttt{; false) handle}\ \mathit{match}\texttt{) handle _ => false} \Rightarrow \mathit{exp}'}{C \vdash \mathit{exp}\ \texttt{raises}\ \mathit{match} \Rightarrow \texttt{(}\mathit{exp}'\texttt{)}} \tag{367}$$

$$\frac{C \vdash exp\ \texttt{raises}\ pat\ \texttt{=>}\ \texttt{true}\ \Rightarrow\ exp'}{C \vdash exp\ \texttt{raises}\ pat \Rightarrow \texttt{(}exp'\texttt{)}} \tag{368}$$

$$\frac{C \vdash \texttt{case}\ exp_1\ \texttt{of}\ \texttt{true}\ \texttt{=>}\ \texttt{(}exp_2\texttt{)}\ \texttt{|}\ \texttt{false}\ \texttt{=>}\ \texttt{(}exp_3\texttt{)}\ \Rightarrow exp}{C \vdash \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3\ \Rightarrow\ \texttt{(}exp\texttt{)}} \tag{369}$$

$$\frac{C \vdash \texttt{(fn}\ match\texttt{)(}exp\texttt{)} \Rightarrow exp'}{C \vdash \texttt{case}\ exp\ \texttt{of}\ match \Rightarrow \texttt{(}exp'\texttt{)}} \tag{370}$$

Comments:

(361) The fixity *fix* can be chosen arbitrarily. If it were fixed, then either left- or right-associative or all operators of precedence 0 would be ruled out.

(362),(363) It is assumed here that `not` has not been rebound, neither in StE nor in VE.

(364)–(369) It is assumed here that `true` and `false` have not been rebound.

Match Rules

$\boxed{C \vdash mrule \Rightarrow mrule'}$

$$\frac{C \vdash pat \Rightarrow pat', StE \qquad C + StE \vdash exp \Rightarrow exp'}{C \vdash pat\ \texttt{=>}\ exp \Rightarrow pat'\ \texttt{=>}\ exp'} \tag{371}$$

Declarations

$\boxed{C \vdash dec \Rightarrow dec', C'}$

$$\frac{C \vdash valbind \Rightarrow valbind', StE}{C \vdash \texttt{val}\ valbind \Rightarrow \texttt{val}\ valbind',\ StE \text{ in Env in Context}} \tag{372}$$

$$\frac{C + StE \vdash fvalbind \Rightarrow valbind, StE}{C \vdash \texttt{fun}\ fvalbind \Rightarrow\ \texttt{val rec}\ valbind,\ StE \text{ in Env in Context}} \tag{373}$$

$$\frac{C \vdash typbind \Rightarrow typbind', TE, TE'}{C \vdash \texttt{type}\ typbind \Rightarrow \texttt{type}\ typbind',\ TE' \text{ in Env in Context}} \tag{374}$$

$$\frac{C \vdash typbind \Rightarrow typbind', TE, TE'}{C \vdash \texttt{eqtype}\ typbind \Rightarrow \texttt{eqtype}\ typbind',\ TE' \text{ in Env in Context}} \tag{375}$$

$$\frac{\begin{array}{c}\langle C + E \vdash typbind^\bullet \Rightarrow typbind', TE, TE'\rangle \\ C + E\langle\ +TE\rangle \vdash datbind \Rightarrow datbind', E\end{array}}{\begin{array}{c}C \vdash \texttt{datatype}\ datbind\ \langle\texttt{withtype}\ typbind^\bullet\rangle \Rightarrow \\ \texttt{datatype}\ datbind'\ \langle\ \texttt{;}\ \texttt{type}\ typbind'\rangle,\ E\langle+TE'\rangle \text{ in Context}\end{array}} \tag{376}$$

$$\frac{\begin{array}{c}\langle C + E \vdash typbind^{\bullet} \Rightarrow typbind', TE, TE' \\ C + E\langle\ +TE\rangle \vdash datbind \Rightarrow datbind', E \\ C + E\langle +TE'\rangle \vdash dec \Rightarrow dec', C'\end{array}}{\begin{array}{c}C \vdash \texttt{abstype}\ datbind\ \langle\texttt{withtype}\ typbind^{\bullet}\rangle\ \texttt{with}\ dec\ \texttt{end} \Rightarrow \\ \texttt{abstype}\ datbind'\ \texttt{with}\ \langle\texttt{type}\ typbind'\ \texttt{;}\rangle\ dec'\ \texttt{end}, \\ \langle TE' \text{ in Context } +\rangle \mathrm{C}'\end{array}} \tag{377}$$

$$\frac{C \vdash exbind \Rightarrow exbind', StE}{C \vdash \texttt{exception}\ exbind \Rightarrow \texttt{exception}\ exbind',\ StE \text{ in Env in Context}} \tag{378}$$

$$\frac{C \vdash dec_1 \Rightarrow dec_1', C_1 \qquad C + C_1 \vdash dec_2 \Rightarrow dec_2', C_2}{C \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow \texttt{local}\ dec_1'\ \texttt{in}\ dec_2'\ \texttt{end},\ C_2} \tag{379}$$

$$\frac{\begin{array}{c}C, SE \text{ of } E \text{ of } _, \texttt{s} \vdash longid_1 \Rightarrow longid_1', E_1 \\ \cdots \\ C, SE \text{ of } E \text{ of } _, \texttt{s} \vdash longid_n \Rightarrow longid_n', E_n\end{array}}{C \vdash \texttt{open}\ longid_1 \cdots longid_n \Rightarrow \texttt{open}\ longid_1' \cdots longid_n', E_1 + \cdots + E_n \text{ in Context}} \tag{380}$$

$$\frac{}{C \vdash \qquad \Rightarrow \qquad , \{\} \text{ in Context}} \tag{381}$$

$$\frac{C \vdash dec_1 \Rightarrow dec_1', C_1 \qquad C + C_1 \vdash dec_2 \Rightarrow dec_2', C_2}{C \vdash dec_1\ \langle;\rangle\ dec_2 \Rightarrow dec_1'\ \langle;\rangle\ dec_2',\ C_1 + C_2} \tag{382}$$

$$\frac{n \geq 1 \qquad VE = \{id \mapsto (0\langle+d\rangle, \texttt{1})\ ;\ id \in \{id_1, \cdots, id_n\}\}}{C \vdash \texttt{infix}\ \langle d\rangle\ id_1 \cdots id_n \Rightarrow \qquad , VE \text{ in Context}} \tag{383}$$

$$\frac{n \geq 1 \qquad VE = \{id \mapsto (0\langle+d\rangle, \texttt{r})\ ;\ id \in \{id_1, \cdots, id_n\}\}}{C \vdash \texttt{infixr}\ \langle d\rangle\ id_1 \cdots id_n \Rightarrow \qquad , VE \text{ in Context}} \tag{384}$$

$$\frac{n \geq 1 \qquad VE = \{id \mapsto (0, \texttt{n})\ ;\ id \in \{id_1, \cdots, id_n\}\}}{C \vdash \texttt{nonfix}\ id_1 \cdots id_n \Rightarrow \qquad , VE \text{ in Context}} \tag{385}$$

Comments:

(376),(377) Type declarations of these forms are mutually recursive.

(383)–(385) Infix directives are replaced by the empty declaration. The digit d is treated in the premise as the corresponding natural number.

Value Bindings

$\boxed{C \vdash \mathit{valbind} \Rightarrow \mathit{valbind}', \mathit{StE}}$

$$\frac{\begin{array}{c} C \vdash \mathit{pat} \Rightarrow \mathit{pat}', \mathit{StE} \qquad C \vdash \mathit{exp} \Rightarrow \mathit{exp}' \\ \langle C \vdash \mathit{valbind} \Rightarrow \mathit{valbind}', \mathit{StE}' \rangle \end{array}}{C \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \Rightarrow \mathit{pat}'\ \texttt{=}\ \mathit{exp}'\ \langle \texttt{and}\ \mathit{valbind}' \rangle,\ \mathit{StE} \langle + \mathit{StE}' \rangle} \tag{386}$$

$$\frac{C + \mathit{StE} \vdash \mathit{valbind} \Rightarrow \mathit{valbind}', \mathit{StE}}{C \vdash \texttt{rec}\ \mathit{valbind} \Rightarrow \texttt{rec}\ \mathit{valbind}', \mathit{StE}} \tag{387}$$

Comments:

(387) The recursive binding of *StE* allows constructor status to be overwritten. This is not possible in SML (although the SML semantics is not entirely clear about this).

Function Value Bindings

$\boxed{C \vdash \mathit{fvalbind} \Rightarrow \mathit{valbind}, \mathit{StE}}$

A function value binding does not make "holes" in the scope of constructors, but we need to make it produce a status environment in order to correctly generate fresh identifiers.

$$\frac{\begin{array}{c} C \vdash \mathit{funcbind} \Rightarrow \mathit{match}, \mathit{var}, n \qquad \mathit{var} = \mathit{id}^{\mathtt{V}} \\ \forall i, j.\ 1 \leq i, j \leq n \wedge i \neq j \implies \mathit{id}_i \neq \mathit{id}_j \\ \forall i.\ 1 \leq i \leq n \implies C, \mathit{StE} \text{ of } E \text{ of } _, \mathtt{v} \vdash \mathit{id}_i \Rightarrow \mathit{var}_i, \text{FAIL} \\ C \vdash \texttt{case}\ (\mathit{id}_1, \cdots, \mathit{id}_n)\ \texttt{of}\ \mathit{match} \Rightarrow \mathit{exp} \\ \langle C \vdash \mathit{fvalbind} \Rightarrow \mathit{valbind}, \mathit{StE} \rangle \end{array}}{\begin{array}{l} C \vdash \mathit{funcbind}\ \langle \texttt{and}\ \mathit{fvalbind} \rangle \Rightarrow \\ \quad \mathit{var}\ \texttt{=}\ \texttt{fn}\ \mathit{var}_1\ \texttt{=>} \cdots \texttt{fn}\ \mathit{var}_n\ \texttt{=>}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle,\ \{\mathit{id} \mapsto \mathtt{v}\} \langle + \mathit{StE} \rangle \end{array}} \tag{388}$$

Single Function Bindings

$\boxed{C \vdash \mathit{funcbind} \Rightarrow \mathit{match}, \mathit{var}, n}$

$$\frac{\begin{array}{c} C \vdash \mathit{fpat} \Rightarrow \mathit{pat}, \mathit{var}, n, \mathit{StE} \qquad \langle C \vdash \mathit{ty} \Rightarrow \mathit{ty}' \rangle \\ C + \mathit{StE} \vdash \mathit{exp} \Rightarrow \mathit{exp}' \qquad \langle \langle C \vdash \mathit{funcbind} \Rightarrow \mathit{match}, \mathit{var}, n \rangle \rangle \end{array}}{\begin{array}{c} C \vdash \mathit{fpat}\ \langle : \mathit{ty} \rangle\ \texttt{=}\ \mathit{exp}\ \langle \langle \texttt{|}\ \mathit{funcbind} \rangle \rangle \Rightarrow \\ \mathit{pat}\ \texttt{=>}\ \mathit{exp}'\ \langle : \mathit{ty}' \rangle\ \langle \langle \texttt{|}\ \mathit{match} \rangle \rangle,\ \mathit{var}, n \end{array}} \tag{389}$$

Comment: Notice that the function name and the number of parameters have to be the same throughout a *funcbind*.

Type Bindings

$\boxed{C \vdash \mathit{typbind} \Rightarrow \mathit{typbind}', \mathit{TE}, \mathit{TE}'}$

A type binding produces two type environments, the first for expansion of type abbreviations (for `withtype` types), the second for declaring the type constructors to be irreducible within type expressions, see rule 423.

$$\frac{tyvarseq = \alpha^{(k)} \qquad C \vdash ty \Rightarrow ty' \qquad \langle C \vdash typbind \Rightarrow typbind', TE, TE' \rangle}{\begin{array}{c} C \vdash tyvarseq\ id\ \texttt{=}\ ty\ \langle \texttt{and}\ typbind \rangle \Rightarrow \\ tyvarseq\ id^{\texttt{t}}\ \texttt{=}\ ty'\ \langle \texttt{and}\ typbind' \rangle, \\ \{id \mapsto \Lambda\alpha^{(k)}.ty'\}\langle +TE \rangle,\ \{id \mapsto \text{FAIL}\}\langle +TE' \rangle \end{array}} \tag{390}$$

$$\frac{tyvarseq = \alpha^{(k)} \qquad \langle C \vdash typbind \Rightarrow typbind', TE, TE' \rangle}{\begin{array}{c} C \vdash tyvarseq\ id\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ typbind \rangle \Rightarrow \\ tyvarseq\ id^{\texttt{t}}\ \texttt{=}\ \texttt{?}\ \langle \texttt{and}\ typbind' \rangle, \\ \{id \mapsto \text{FAIL}\}\langle +TE \rangle,\ \{id \mapsto \text{FAIL}\}\langle +TE' \rangle \end{array}} \tag{391}$$

Data Type Bindings

$\boxed{C \vdash datbind \Rightarrow datbind', E}$

$$\frac{C \vdash conbind \Rightarrow conbind', StE \qquad \langle C \vdash datbind \Rightarrow datbind', E \rangle}{\begin{array}{c} C \vdash tyvarseq\ id\ \texttt{=}\ conbind\ \langle \texttt{and}\ datbind \rangle \Rightarrow \\ tyvarseq\ id^{\texttt{t}}\ \texttt{=}\ conbind'\ \langle \texttt{and}\ datbind' \rangle, \\ (\{id \mapsto \text{FAIL}\}, StE)\ \text{in Env}\ \langle +E \rangle \end{array}} \tag{392}$$

Constructor Bindings

$\boxed{C \vdash conbind \Rightarrow conbind', StE}$

$$\frac{\langle C \vdash conbind \Rightarrow conbind', StE \rangle}{C \vdash id\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow id^{\texttt{c}}\ \langle\ \texttt{|}\ conbind' \rangle,\ \{id \mapsto \texttt{c}\}\ \langle +StE \rangle} \tag{393}$$

$$\frac{C \vdash ty \Rightarrow ty' \qquad \langle C \vdash conbind \Rightarrow conbind', StE \rangle}{C \vdash id\ \texttt{of}\ ty\ \langle\ \texttt{|}\ conbind \rangle \Rightarrow id^{\texttt{c}}\ \texttt{of}\ ty'\ \langle\ \texttt{|}\ conbind' \rangle,\ \{id \mapsto \texttt{c}\}\ \langle +StE \rangle} \tag{394}$$

Exception Bindings

$\boxed{C \vdash exbind \Rightarrow exbind', StE}$

$$\frac{\langle C \vdash ty \Rightarrow ty' \rangle \qquad \langle\langle C \vdash exbind \Rightarrow exbind', StE \rangle\rangle}{\begin{array}{c} C \vdash id\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exbind \rangle\rangle \Rightarrow \\ id^{\texttt{e}}\ \langle \texttt{of}\ ty' \rangle\ \langle\langle \texttt{and}\ exbind' \rangle\rangle,\ \{id \mapsto \texttt{e}\}\langle\langle +StE \rangle\rangle \end{array}} \tag{395}$$

$$\frac{\begin{array}{c} C, StE\ \text{of}\ E\ \text{of}\ _, \texttt{e} \vdash longid \Rightarrow longexcon, \texttt{e} \\ \langle C \vdash exbind \Rightarrow exbind', StE \rangle \end{array}}{\begin{array}{c} C \vdash id\ \texttt{=}\ longid\ \langle \texttt{and}\ exbind \rangle \Rightarrow \\ id^{\texttt{e}}\ \texttt{=}\ longexcon\ \langle \texttt{and}\ exbind' \rangle,\ \{id \mapsto \texttt{e}\}\langle +StE \rangle \end{array}} \tag{396}$$

Atomic Patterns

$$\boxed{C \vdash atpat \Rightarrow atpat', StE}$$

$$\frac{}{C \vdash _ \Rightarrow _, \{\}} \tag{397}$$

$$\frac{}{C \vdash scon \Rightarrow scon, \{\}} \tag{398}$$

$$\frac{C \vdash \langle \texttt{op} \rangle\ id \Rightarrow var, \texttt{v}}{C \vdash \langle \texttt{op} \rangle\ id \Rightarrow var, \{id \mapsto \texttt{v}\}} \tag{399}$$

$$\frac{C \vdash \langle \texttt{op} \rangle\ longid \Rightarrow longid', st \quad st \neq \texttt{v}}{C \vdash \langle \texttt{op} \rangle\ longid \Rightarrow longid', \{\}} \tag{400}$$

$$\frac{\langle C \vdash patrow \Rightarrow patrow', StE \rangle}{C \vdash \{\ \langle patrow \rangle\ \} \Rightarrow \{\ \langle patrow' \rangle\ \},\ \{\} \langle + StE \rangle} \tag{401}$$

$$\frac{}{C \vdash \texttt{()} \Rightarrow \{\},\ \{\}} \tag{402}$$

$$\frac{n \geq 2 \qquad C \vdash pat_1 \Rightarrow pat'_1, StE_1 \quad \cdots \quad C \vdash pat_n \Rightarrow pat'_n, StE_n}{C \vdash \texttt{(}pat_1\texttt{,}\cdots\texttt{,}pat_n\texttt{)} \Rightarrow \{\texttt{1=}pat_1\texttt{,}\cdots\texttt{,}\overline{n}\texttt{=}pat_n\},\ StE_1 + \cdots + StE_n} \tag{403}$$

$$\frac{n \geq 0 \qquad C \vdash \texttt{(}pat_1\texttt{)::}\cdots\texttt{::(}pat_n\texttt{)::nil} \Rightarrow pat, StE}{C \vdash \texttt{[}pat_1\texttt{,}\cdots\texttt{,}pat_n\texttt{]} \Rightarrow \texttt{(}pat\texttt{)}, StE} \tag{404}$$

$$\frac{C \vdash pat \Rightarrow pat', StE}{C \vdash \texttt{(}pat\texttt{)} \Rightarrow \texttt{(}pat'\texttt{)}, StE} \tag{405}$$

Comments:

(404) It is assumed here that neither `nil` nor `::` has been rebound, and `::` still has right-associative infix status.

Pattern Rows

$$\boxed{C \vdash patrow \Rightarrow patrow', StE}$$

$$\frac{}{C \vdash \ldots \Rightarrow \ldots, \{\}} \tag{406}$$

$$\frac{C \vdash pat \Rightarrow pat', StE \qquad \langle C \vdash patrow \Rightarrow patrow', StE' \rangle}{C \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow \rangle \Rightarrow lab\ \texttt{=}\ pat'\ \langle\ \texttt{,}\ patrow' \rangle,\ StE\ \langle + StE' \rangle} \tag{407}$$

$$\frac{lab = id \text{ in Lab} \qquad C \vdash lab\ \texttt{=}\ id \langle \texttt{:}\ ty \rangle\ \langle \langle\ \texttt{,}\ patrow \rangle \rangle \Rightarrow patrow', StE}{C \vdash id \langle \texttt{:}\ ty \rangle\ \langle \langle\ \texttt{,}\ patrow \rangle \rangle \Rightarrow patrow', StE} \tag{408}$$

$$\frac{lab = id \text{ in Lab} \qquad C \vdash lab\ \texttt{=}\ id \langle \texttt{:}\ ty \rangle\ \texttt{as}\ pat\ \langle \langle\ \texttt{,}\ patrow \rangle \rangle \Rightarrow patrow', StE}{C \vdash id \langle \texttt{:}\ ty \rangle\ \texttt{as}\ pat\ \langle \langle\ \texttt{,}\ patrow \rangle \rangle \Rightarrow patrow', StE} \tag{409}$$

Application Patterns

$$\boxed{C \vdash \mathit{apppat} \Rightarrow \mathit{pat}, \mathit{StE}}$$

$$\frac{C \vdash \langle \mathtt{op} \rangle \mathit{longid} \Rightarrow \mathit{longid}', \mathit{st} \quad \mathit{st} \neq \mathtt{v} \qquad C \vdash \mathit{atpat} \Rightarrow \mathit{atpat}', \mathit{StE}}{C \vdash \langle \mathtt{op} \rangle \mathit{longid}\ \ \mathit{atpat} \Rightarrow \mathit{longid}'\ \ \mathit{atpat}', \mathit{StE}} \tag{410}$$

$$\frac{C \vdash \mathit{atpat} \Rightarrow \mathit{atpat}', \mathit{StE}}{C \vdash \mathit{atpat} \Rightarrow \mathit{atpat}', \mathit{StE}} \tag{411}$$

Infix Patterns

$$\boxed{C, n, \mathit{fix} \vdash \mathit{infpat} \Rightarrow \mathit{pat}, \mathit{StE}}$$

$$\frac{C \vdash \mathit{apppat} \Rightarrow \mathit{pat}, \mathit{StE}}{C, n, \mathit{fix} \vdash \mathit{apppat} \Rightarrow \mathit{pat}, \mathit{StE}} \tag{412}$$

$$\frac{\begin{array}{c}[\mathit{VE}\text{ of }C](\mathit{id}) = (n', \mathtt{l}) \qquad n < n' \vee (n = n' \wedge \mathit{fix} = \mathtt{l}) \\ [\mathit{StE}\text{ of }E\text{ of }C](\mathit{id}) = \mathit{st} \in \{\mathtt{c}, \mathtt{e}\} \\ C, n', \mathtt{l} \vdash \mathit{infpat}_1 \Rightarrow \mathit{pat}_1, \mathit{StE}_1 \qquad C, n', \mathtt{n} \vdash \mathit{infpat}_2 \Rightarrow \mathit{pat}_2, \mathit{StE}_2\end{array}}{C, n, \mathit{fix} \vdash \mathit{infpat}_1\ \mathit{id}\ \mathit{infpat}_2 \Rightarrow \mathit{id}^{\mathit{st}}\ \{\mathtt{1=}\mathit{pat}_1\mathtt{,2=}\mathit{pat}_2\},\ \mathit{StE}_1 + \mathit{StE}_2} \tag{413}$$

$$\frac{\begin{array}{c}[\mathit{VE}\text{ of }C](\mathit{id}) = (n', \mathtt{r}) \qquad n < n' \vee (n = n' \wedge \mathit{fix} = \mathtt{r}) \\ [\mathit{StE}\text{ of }E\text{ of }C](\mathit{id}) = \mathit{st} \in \{\mathtt{c}, \mathtt{e}\} \\ C, n', \mathtt{n} \vdash \mathit{infpat}_1 \Rightarrow \mathit{pat}_1, \mathit{StE}_1 \qquad C, n', \mathtt{r} \vdash \mathit{infpat}_2 \Rightarrow \mathit{pat}_2, \mathit{StE}_2\end{array}}{C, n, \mathit{fix} \vdash \mathit{infpat}_1\ \mathit{id}\ \mathit{infpat}_2 \Rightarrow \mathit{id}^{\mathit{st}}\ \{\mathtt{1=}\mathit{pat}_1\mathtt{,2=}\mathit{pat}_2\},\ \mathit{StE}_1 + \mathit{StE}_2} \tag{414}$$

Patterns

$$\boxed{C \vdash \mathit{pat} \Rightarrow \mathit{pat}', \mathit{StE}}$$

$$\frac{C, 0, \mathit{fix} \vdash \mathit{infpat} \Rightarrow \mathit{pat}, \mathit{StE}}{C \vdash \mathit{infpat} \Rightarrow \mathit{pat}, \mathit{StE}} \tag{415}$$

$$\frac{C \vdash \mathit{ty} \Rightarrow \mathit{ty}' \qquad C \vdash \mathit{pat} \Rightarrow \mathit{pat}', \mathit{StE}}{C \vdash \mathit{pat} : \mathit{ty} \Rightarrow \mathit{pat}' : \mathit{ty}', \mathit{StE}} \tag{416}$$

$$\frac{C \vdash \langle \mathtt{op} \rangle \mathit{id} \Rightarrow \mathit{var}, \mathtt{v} \qquad \langle\langle C \vdash \mathit{ty} \Rightarrow \mathit{ty}' \rangle\rangle \qquad C \vdash \mathit{pat} \Rightarrow \mathit{pat}', \mathit{StE}}{C \vdash \langle \mathtt{op} \rangle \mathit{id} \langle\langle : \mathit{ty} \rangle\rangle\ \mathtt{as}\ \mathit{pat} \Rightarrow \mathit{var} \langle\langle : \mathit{ty}' \rangle\rangle\ \mathtt{as}\ \mathit{pat}', \{\mathit{id} \mapsto \mathtt{v}\} + \mathit{StE}} \tag{417}$$

Function Patterns

$$\boxed{C \vdash \mathit{fpat} \Rightarrow \mathit{pat}, \mathit{var}, n, \mathit{StE}}$$

$$\frac{n \geq 1 \qquad C \vdash (\mathit{atpat}_1, \cdots, \mathit{atpat}_n) \Rightarrow \mathit{pat}, \mathit{StE}}{C \vdash \mathtt{op}\ \mathit{id}\ \mathit{atpat}_1 \cdots \mathit{atpat}_n \Rightarrow \mathit{pat}, \mathit{id}^{\mathtt{V}}, n, \mathit{StE}} \tag{418}$$

$$\frac{n \geq 1 \qquad [\mathit{VE}\text{ of }C](\mathit{id}) = (n', \mathtt{n}) \qquad C \vdash (\mathit{atpat}_1, \cdots, \mathit{atpat}_n) \Rightarrow \mathit{pat}, \mathit{StE}}{C \vdash \mathit{id}\ \mathit{atpat}_1 \cdots \mathit{atpat}_n \Rightarrow \mathit{pat}, \mathit{id}^{\mathtt{V}}, n, \mathit{StE}} \tag{419}$$

$$\frac{\begin{array}{c}C \vdash ((atpat_1\,,atpat'_1)\,,atpat_2\,,\cdots,atpat_n) \Rightarrow pat, StE \\ [VE \text{ of } C](id) = (n', fix) \quad fix \neq \texttt{n} \qquad n \geq 1\end{array}}{C \vdash (atpat_1\ id\ atpat'_1)\ atpat_2 \cdots atpat_n \Rightarrow pat, id^{\texttt{V}}, n, StE} \tag{420}$$

$$\frac{[VE \text{ of } C](id) = (n', fix) \quad fix \neq \texttt{n} \qquad C \vdash (atpat_1\,,atpat'_1) \Rightarrow pat, StE}{C \vdash atpat_1\ id\ atpat'_1 \Rightarrow pat, id^{\texttt{V}}, 1, StE} \tag{421}$$

Type Expressions

$\boxed{C \vdash ty \Rightarrow ty'}$

$$\frac{\begin{array}{c}tyseq = ty_1 \cdots ty_k \qquad C \vdash ty_1 \Rightarrow ty'_1 \quad \cdots \quad C \vdash ty_k \Rightarrow ty'_k \\ C, TE \text{ of } E \text{ of } _, \texttt{t} \vdash longid \Rightarrow longtycon, \Lambda\alpha_1 \cdots \alpha_k.ty\end{array}}{C \vdash tyseq\ longid \Rightarrow ty\{ty'_1/\alpha_1, \ldots, ty'_k/\alpha_k\}} \tag{422}$$

$$\frac{\begin{array}{c}C, TE \text{ of } E \text{ of } _, \texttt{t} \vdash longid \Rightarrow longtycon, \text{FAIL} \qquad tyseq = ty_1 \cdots ty_k \\ C \vdash ty_1 \Rightarrow ty'_1 \quad \cdots \quad C \vdash ty_k \Rightarrow ty'_k \qquad tyseq' = ty'_1 \cdots ty'_k\end{array}}{C \vdash tyseq\ longid \Rightarrow tyseq'\ longtycon} \tag{423}$$

$$\frac{n \geq 2 \qquad C \vdash \{\texttt{1}: ty_1\,,\cdots,\overline{n}: ty_n\} \Rightarrow ty}{C \vdash ty_1 \texttt{*} \cdots \texttt{*} ty_n \Rightarrow ty} \tag{424}$$

Comments:

(422) This rule replaces type constructors defined in a `withtype` clause of a type declaration. The replacement of `withtype` types of the current context C does not apply to type constructors in ty — ty is a type expression for which `withtype` types have already been replaced.

Structure Expressions

$\boxed{C \vdash strexp \Rightarrow strexp', E}$

$$\frac{C \vdash strdec \Rightarrow strdec', C'}{C \vdash \texttt{struct}\ strdec\ \texttt{end} \Rightarrow \texttt{struct}\ strdec'\ \texttt{end}, E \text{ of } C'} \tag{425}$$

$$\frac{C, SE \text{ of } E \text{ of } _, \texttt{s} \vdash longid \Rightarrow longstrid, E}{C \vdash longid \Rightarrow longstrid, E} \tag{426}$$

$$\frac{C, F \text{ of } _, \texttt{f} \vdash id \Rightarrow funid, (E_1, E_2) \qquad C \vdash strexp \Rightarrow strexp', E \quad E \succ E_1}{C \vdash id\ (\ strexp\ ) \vdash funid\ (\ strexp'\ ),\ E_2} \tag{427}$$

$$\frac{C \vdash id\ (\ \texttt{struct}\ strdec\ \texttt{end}\ ) \Rightarrow strexp', E}{C \vdash id\ (\ strdec\ ) \Rightarrow strexp', E} \tag{428}$$

$$\frac{C \vdash strdec \Rightarrow strdec', C' \qquad C + C' \vdash strexp \Rightarrow strexp', E}{C \vdash \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end} \Rightarrow \texttt{let}\ strdec'\ \texttt{in}\ strexp'\ \texttt{end},\ E} \tag{429}$$

Structure-level Declarations $\boxed{C \vdash strdec \Rightarrow strdec', C'}$

$$\frac{C \vdash dec \Rightarrow dec', C'}{C \vdash dec \Rightarrow dec', C'} \tag{430}$$

$$\frac{C \vdash ax \Rightarrow ax'}{C \vdash \texttt{axiom}\ ax \Rightarrow \texttt{axiom}\ ax',\ \{\}\ \text{in Context}} \tag{431}$$

$$\frac{C \vdash strbind \Rightarrow strbind', SE}{C \vdash \texttt{structure}\ strbind \Rightarrow \texttt{structure}\ strbind',\ SE\ \text{in Env in Context}} \tag{432}$$

$$\frac{C \vdash strdec_1 \Rightarrow strdec_1', C_1 \qquad C + C_1 \vdash strdec_2 \Rightarrow strdec_2', C_2}{C \vdash \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \Rightarrow \texttt{local}\ strdec_1'\ \texttt{in}\ strdec_2'\ \texttt{end},\ C_2} \tag{433}$$

$$\frac{}{C \vdash \quad \Rightarrow \quad, \{\}\ \text{in Context}} \tag{434}$$

$$\frac{C \vdash strdec_1 \Rightarrow strdec_1', C_1 \qquad C + C_1 \vdash strdec_2 \Rightarrow strdec_2', C_2}{C \vdash strdec_1\ \langle ; \rangle\ strdec_2 \Rightarrow strdec_1'\ \langle ; \rangle\ strdec_2',\ C_1 + C_2} \tag{435}$$

Structure Bindings $\boxed{C \vdash strbind \Rightarrow SE}$

$$\frac{C \vdash sglstrbind \Rightarrow sglstrbind', SE_1 \qquad \langle C \vdash strbind \Rightarrow strbind', SE_2 \rangle}{C \vdash sglstrbind\ \langle \texttt{and}\ strbind \rangle \Rightarrow sglstrbind'\ \langle \texttt{and}\ strbind' \rangle,\ SE_1\ \langle + SE_2 \rangle} \tag{436}$$

Single Structure Bindings $\boxed{C \vdash sglstrbind \Rightarrow sglstrbind', SE}$

$$\frac{C \vdash psigexp \Rightarrow psigexp', E_1 \qquad C \vdash strexp \Rightarrow strexp', E_2 \qquad E_2 \succ E_1}{C \vdash id : psigexp\ \texttt{=}\ strexp \Rightarrow id^{\texttt{S}} : psigexp'\ \texttt{=}\ strexp',\ \{id \mapsto E_1\}} \tag{437}$$

$$\frac{C \vdash psigexp \Rightarrow psigexp', E}{C \vdash id : psigexp\ \texttt{=}\ \texttt{?} \Rightarrow id^{\texttt{S}} : psigexp'\ \texttt{=}\ \texttt{?},\ \{id \mapsto E\}} \tag{438}$$

$$\frac{C \vdash strexp \Rightarrow strexp', E}{C \vdash id\ \texttt{=}\ strexp \Rightarrow id^{\texttt{S}}\ \texttt{=}\ strexp',\ \{id \mapsto E\}} \tag{439}$$

Signature Expressions $\boxed{C \vdash sigexp \Rightarrow sigexp', E}$

$$\frac{C \vdash spec \Rightarrow spec', E}{C \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow \texttt{sig}\ spec'\ \texttt{end}, E} \tag{440}$$

$$\frac{C, G\ \text{of}\ _, \mathsf{g} \vdash id \Rightarrow sigid, E}{C \vdash id \Rightarrow sigid, E} \tag{441}$$

Principal Signatures

$\boxed{C \vdash psigexp \Rightarrow psigexp', E}$

$$\frac{C \vdash sigexp \Rightarrow sigexp', E}{C \vdash sigexp \Rightarrow sigexp', E} \tag{442}$$

Signature Declarations

$\boxed{C \vdash sigdec \Rightarrow sigdec', G}$

$$\frac{C \vdash sigbind \Rightarrow sigbind', G}{C \vdash \texttt{signature}\ sigbind \Rightarrow \texttt{signature}\ sigbind', G} \tag{443}$$

$$\frac{}{C \vdash \quad \Rightarrow \quad , \{\}} \tag{444}$$

$$\frac{C \vdash sigdec_1 \Rightarrow sigdec_1', G_1 \qquad C + G_1 \vdash sigdec_2 \Rightarrow sigdec_2', G_2}{C \vdash sigdec_1\ \langle ; \rangle\ sigdec_2 \Rightarrow sigdec_1'\ \langle ; \rangle\ sigdec_2',\ G_1 + G_2} \tag{445}$$

Signature Bindings

$\boxed{C \vdash sigbind \Rightarrow sigbind', G}$

$$\frac{C \vdash psigexp \Rightarrow psigexp', E \qquad \langle C \vdash sigbind \Rightarrow sigbind', G \rangle}{\begin{array}{l} C \vdash id\ \texttt{=}\ psigexp\ \langle \texttt{and}\ sigbind \rangle \Rightarrow \\ \qquad id^{\mathsf{g}}\ \texttt{=}\ psigexp'\ \langle \texttt{and}\ sigbind' \rangle,\ \{id \mapsto E\}\ \langle +G \rangle \end{array}} \tag{446}$$

Specifications

$\boxed{C \vdash spec \Rightarrow spec', E}$

$$\frac{C \vdash valdesc \Rightarrow valdesc', E}{C \vdash \texttt{val}\ valdesc \Rightarrow \texttt{val}\ valdesc', E} \tag{447}$$

$$\frac{C \vdash typdesc \Rightarrow typdesc', TE}{C \vdash \texttt{type}\ typdesc \Rightarrow \texttt{type}\ typdesc', TE \text{ in Env}} \tag{448}$$

$$\frac{C \vdash typdesc \Rightarrow typdesc', TE}{C \vdash \texttt{eqtype}\ typdesc \Rightarrow \texttt{eqtype}\ typdesc', TE \text{ in Env}} \tag{449}$$

$$\frac{C \vdash datdesc \Rightarrow datdesc', E}{C \vdash \texttt{datatype}\ datdesc \Rightarrow \texttt{datatype}\ datdesc',\ E} \tag{450}$$

$$\frac{C \vdash exdesc \Rightarrow exdesc', StE}{C \vdash \texttt{exception}\ exdesc \Rightarrow \texttt{exception}\ exdesc',\ StE \text{ in Env}} \tag{451}$$

$$\frac{C \vdash axdesc \Rightarrow axdesc'}{C \vdash \texttt{axiom}\ axdesc \Rightarrow \texttt{axiom}\ axdesc',\ \{\} \text{ in Env}} \tag{452}$$

$$\frac{C \vdash strdesc \Rightarrow strdesc', SE}{C \vdash \texttt{structure}\ strdesc \Rightarrow \texttt{structure}\ strdesc',\ SE \text{ in Env}} \tag{453}$$

$$\frac{C \vdash shareq \Rightarrow shareq'}{C \vdash \texttt{sharing}\ shareq \Rightarrow \texttt{sharing}\ shareq',\ \{\}\ \text{in Env}} \tag{454}$$

$$\frac{C \vdash spec_1 \Rightarrow spec_1', E_1 \qquad C + E_1 \vdash spec_2 \Rightarrow spec_2', E_2}{C \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow \texttt{local}\ spec_1'\ \texttt{in}\ spec_2'\ \texttt{end},\ E_2} \tag{455}$$

$$\frac{\begin{array}{c} C, SE\ \text{of}\ E\ \text{of}\ _, \texttt{s} \vdash longid_1 \Rightarrow longstrid_1, E_1 \\ \cdots \\ C, SE\ \text{of}\ E\ \text{of}\ _, \texttt{s} \vdash longid_n \Rightarrow longstrid_n, E_n \end{array}}{\begin{array}{l} C \vdash \texttt{open}\ longid_1 \cdots longid_n \Rightarrow \\ \qquad \texttt{open}\ longstrid_1 \cdots longstrid_n,\ E_1 + \cdots + E_n \end{array}} \tag{456}$$

$$\frac{\begin{array}{c} C, G\ \text{of}\ _, \texttt{g} \vdash id_1 \Rightarrow sigid_1, E_1 \\ \cdots \\ C, G\ \text{of}\ _, \texttt{g} \vdash id_n \Rightarrow sigid_n, E_n \end{array}}{C \vdash \texttt{include}\ id_1 \cdots id_n \Rightarrow \texttt{include}\ sigid_1 \cdots sigid_n,\ E_1 + \cdots + E_n} \tag{457}$$

$$\frac{}{C \vdash \qquad \Rightarrow \qquad, \{\}\ \text{in Env}} \tag{458}$$

$$\frac{C \vdash spec_1 \Rightarrow spec_1', E_1 \qquad C + E_1 \vdash spec_2 \Rightarrow spec_2', E_2}{C \vdash spec_1\ \langle ; \rangle\ spec_2 \Rightarrow spec_1'\ \langle ; \rangle\ spec_2',\ E_1 + E_2} \tag{459}$$

Value Descriptions

$\boxed{C \vdash valdesc \Rightarrow valdesc', StE}$

$$\frac{C \vdash ty \Rightarrow ty' \qquad \langle C \vdash valdesc \Rightarrow valdesc', StE \rangle}{C \vdash id\ :\ ty\ \langle \texttt{and}\ valdesc \rangle \Rightarrow id^{\texttt{v}}\ :\ ty'\ \langle \texttt{and}\ valdesc' \rangle,\ \{id \mapsto \texttt{v}\} \langle + StE \rangle} \tag{460}$$

Type Descriptions

$\boxed{C \vdash typdesc \Rightarrow typdesc', TE}$

$$\frac{\langle C \vdash typdesc \Rightarrow typdesc', TE \rangle}{\begin{array}{l} C \vdash tyvarseq\ id\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \\ \qquad tyvarseq\ id^{\texttt{t}}\ \langle \texttt{and}\ typdesc' \rangle,\ \{id \mapsto \text{FAIL}\} \langle + TE \rangle \end{array}} \tag{461}$$

Datatype Description

$\boxed{C \vdash datdesc \Rightarrow datdesc', E}$

$$\frac{C \vdash condesc \Rightarrow condesc', StE \qquad \langle C \vdash datdesc \Rightarrow datdesc', E \rangle}{\begin{array}{c} C \vdash tyvarseq\ id\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ tyvarseq\ id^{\texttt{t}}\ \texttt{=}\ condesc'\ \langle \texttt{and}\ datdesc' \rangle, \\ (\{id \mapsto \text{FAIL}\}, StE)\ \text{in Env}\ \langle + E \rangle \end{array}} \tag{462}$$

Constructor Descriptions $\boxed{C \vdash condesc \Rightarrow condesc', StE}$

$$\frac{\langle C \vdash ty \Rightarrow ty' \rangle \qquad \langle\langle C \vdash condesc \Rightarrow condesc', StE \rangle\rangle}{C \vdash id\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{|}\ condesc \rangle\rangle \Rightarrow \\ id^{\texttt{c}}\ \langle \texttt{of}\ ty' \rangle\ \langle\langle \texttt{|}\ condesc' \rangle\rangle,\quad \{id \mapsto \texttt{c}\}\langle\langle +StE \rangle\rangle} \tag{463}$$

Exception Descriptions $\boxed{C \vdash exdesc \Rightarrow exdesc', StE}$

$$\frac{\langle C \vdash ty \Rightarrow ty' \rangle \qquad \langle\langle C \vdash exdesc \Rightarrow exdesc', StE \rangle\rangle}{C \vdash id\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exdesc \rangle\rangle \Rightarrow \\ id^{\texttt{e}}\ \langle \texttt{of}\ ty' \rangle\ \langle\langle \texttt{and}\ exdesc' \rangle\rangle,\ \{id \mapsto \texttt{e}\}\ \langle\langle +StE \rangle\rangle} \tag{464}$$

Specification Expressions $\boxed{C \vdash specexp \Rightarrow specexp'}$

$$\frac{C \vdash strdec \Rightarrow strdec', E \qquad C + E \vdash axexp \Rightarrow exp}{C \vdash \texttt{let}\ strdec\ \texttt{in}\ axexp\ \texttt{end} \Rightarrow \texttt{let}\ strdec'\ \texttt{in}\ exp\ \texttt{end}} \tag{465}$$

$$\frac{C \vdash \texttt{let}\quad \texttt{in}\ axexp\ \texttt{end} \Rightarrow specexp}{C \vdash axexp \Rightarrow specexp} \tag{466}$$

Axiomatic Expressions $\boxed{C \vdash axexp \Rightarrow exp}$

$$\frac{C \vdash exp^{\bullet} \Rightarrow exp}{C \vdash exp^{\bullet} \Rightarrow exp} \tag{467}$$

Structure Descriptions $\boxed{C \vdash strdesc \Rightarrow strdesc', SE}$

$$\frac{C \vdash sigexp \Rightarrow sigexp', E \qquad \langle C \vdash strdesc \Rightarrow strdesc', SE \rangle}{C \vdash id\ :\ sigexp\ \langle \texttt{and}\ strdesc \rangle \Rightarrow \\ id^{\texttt{s}}\ :\ sigexp'\ \langle \texttt{and}\ strdesc' \rangle,\ \{id \mapsto E\}\ \langle +SE \rangle} \tag{468}$$

Functor Declarations $\boxed{C \vdash fundec \Rightarrow fundec', F}$

$$\frac{C \vdash funbind \Rightarrow funbind', F}{C \vdash \texttt{functor}\ funbind \Rightarrow \texttt{functor}\ funbind', F} \tag{469}$$

$$\frac{}{C \vdash \quad \Rightarrow \quad, \{\}} \tag{470}$$

$$\frac{C \vdash fundec_1 \Rightarrow fundec_1', F_1 \qquad C + F_1 \vdash fundec_2 \Rightarrow fundec_2', F_2}{C \vdash fundec_1\ \langle ; \rangle\ fundec_2 \Rightarrow fundec_1'\ \langle ; \rangle\ fundec_2',\ F_1 + F_2} \tag{471}$$

Functor Bindings

$\boxed{C \vdash \mathit{funbind} \Rightarrow \mathit{funbind}', F}$

$$\frac{\begin{array}{c} C, \mathit{SE} \text{ of } E \text{ of } _, \mathtt{s} \vdash \mathit{id}'' \Rightarrow \mathit{strid}, \text{FAIL} \\ C \vdash \mathit{id}\ \mathtt{(}\ \mathit{id}''\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}\ \mathtt{end}\ \mathtt{)}\ \mathtt{:}\ \mathit{id}'\ \mathtt{=} \\ \mathtt{let\ open}\ \mathit{id}''\ \mathtt{in}\ \mathit{strexp}\ \mathtt{end}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F \end{array}}{C \vdash \mathit{id}\ \mathtt{(}\ \mathit{spec}\ \mathtt{)}\ \mathtt{:}\ \mathit{id}'\ \mathtt{=}\ \mathit{strexp}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F} \tag{472}$$

$$\frac{\begin{array}{c} C, \mathit{SE} \text{ of } E \text{ of } _, \mathtt{s} \vdash \mathit{id}' \Rightarrow \mathit{strid}, \text{FAIL} \\ C \vdash \mathit{id}\ \mathtt{(}\ \mathit{id}'\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}\ \mathtt{end}\ \mathtt{)}\ \mathtt{:}\ \mathtt{sig\ local\ open}\ \mathit{id}'\ \mathtt{in}\ \mathit{spec}'\ \mathtt{end\ end} \\ \mathtt{=\ let\ open}\ \mathit{id}'\ \mathtt{in}\ \mathit{strexp}\ \mathtt{end}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F \end{array}}{C \vdash \mathit{id}\ \mathtt{(}\ \mathit{spec}\ \mathtt{)}\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}'\ \mathtt{end}\ \mathtt{=}\ \mathit{strexp}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F} \tag{473}$$

$$\frac{C, \mathit{SE} \text{ of } E \text{ of } _, \mathtt{s} \vdash \mathit{id}'' \Rightarrow \mathit{strid}, \text{FAIL} \quad C \vdash \mathit{id}\ \mathtt{(}\ \mathit{id}''\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}\ \mathtt{end}\ \mathtt{)}\ \mathtt{:}\ \mathit{id}'\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F}{C \vdash \mathit{id}\ \mathtt{(}\ \mathit{spec}\ \mathtt{)}\ \mathtt{:}\ \mathit{id}'\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F} \tag{474}$$

$$\frac{\begin{array}{c} C, \mathit{SE} \text{ of } E \text{ of } _, \mathtt{s} \vdash \mathit{id}' \Rightarrow \mathit{strid}, \text{FAIL} \\ C \vdash \mathit{id}\ \mathtt{(}\ \mathit{id}'\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}\ \mathtt{end}\ \mathtt{)}\ \mathtt{:}\ \mathtt{sig\ local\ open}\ \mathit{id}'\ \mathtt{in}\ \mathit{spec}'\ \mathtt{end\ end} \\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F \end{array}}{C \vdash \mathit{id}\ \mathtt{(}\ \mathit{spec}\ \mathtt{)}\ \mathtt{:}\ \mathtt{sig}\ \mathit{spec}'\ \mathtt{end}\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \mathit{funbind}', F} \tag{475}$$

$$\frac{\begin{array}{c} C \vdash \mathit{psigexp}_1 \Rightarrow \mathit{psigexp}_1', E_1 \\ C + (\{\mathit{id}_2 \mapsto E_1\} \text{ in Env}) \vdash \mathit{psigexp}_2 \Rightarrow \mathit{psigexp}_2', E_2 \\ C + (\{\mathit{id}_2 \mapsto E_1\} \text{ in Env}) \vdash \mathit{strexp} \Rightarrow \mathit{strexp}', E \qquad E \succ E_2 \\ \langle C \vdash \mathit{funbind} \Rightarrow \mathit{funbind}', F\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{id}_1\ \mathtt{(}\ \mathit{id}_2\ \mathtt{:}\ \mathit{psigexp}_1\ \mathtt{)}\ \mathtt{:}\mathit{psigexp}_2\ \mathtt{=}\ \mathit{strexp}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \mathit{id}_1^{\mathtt{f}}\ \mathtt{(}\ \mathit{id}_2^{\mathtt{s}}\ \mathtt{:}\ \mathit{psigexp}_1'\ \mathtt{)}\ \mathtt{:}\mathit{psigexp}_2'\ \mathtt{=}\ \mathit{strexp}'\ \langle \mathtt{and}\ \mathit{funbind}'\rangle, \\ \{\mathit{funid} \mapsto (E_1, E_2)\}\ \langle + F\rangle \end{array}} \tag{476}$$

$$\frac{\begin{array}{c} C \vdash \mathit{psigexp}_1 \Rightarrow \mathit{psigexp}_1', E_1 \\ C + (\{\mathit{id}_2 \mapsto E_1\} \text{ in Env}) \vdash \mathit{psigexp}_2 \Rightarrow \mathit{psigexp}_2', E_2 \\ \langle C \vdash \mathit{funbind} \Rightarrow \mathit{funbind}', F\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{id}_1\ \mathtt{(}\ \mathit{id}_2\ \mathtt{:}\ \mathit{psigexp}_1\ \mathtt{)}\ \mathtt{:}\mathit{psigexp}_2\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \mathit{id}_1^{\mathtt{f}}\ \mathtt{(}\ \mathit{id}_2^{\mathtt{s}}\ \mathtt{:}\ \mathit{psigexp}_1'\ \mathtt{)}\ \mathtt{:}\mathit{psigexp}_2'\ \mathtt{=\ ?}\ \langle \mathtt{and}\ \mathit{funbind}'\rangle, \\ \{\mathit{funid} \mapsto (E_1, E_2)\}\ \langle + F\rangle \end{array}} \tag{477}$$

Top-level Declarations

$\boxed{C \vdash topdec \Rightarrow topdec', C'}$

$$\frac{C \vdash \texttt{val it =}\ exp \Rightarrow topdec, C'}{C \vdash exp \Rightarrow topdec, C'} \quad (478)$$

$$\frac{C \vdash strdec \Rightarrow strdec', C'}{C \vdash strdec \Rightarrow strdec', C'} \quad (479)$$

$$\frac{C \vdash sigdec \Rightarrow sigdec', G}{C \vdash sigdec \Rightarrow sigdec', G \text{ in Context}} \quad (480)$$

$$\frac{C \vdash fundec \Rightarrow fundec', F}{C \vdash fundec \Rightarrow fundec', F \text{ in Context}} \quad (481)$$

C Appendix: The Initial Basis

For the verification of full programs, see rule 343, we have to specify the initial (verification) basis B_0, an initial context C^0_{DER} for deriving phrases of the Bare language from phrases of the Full language, and an initial (verification) state s_0. The subscript "DER" refers to the semantics of derived forms in Appendix B.

C.1 The Initial State

The initial state s_0 has the form

$$s_0 \;=\; (\text{BasExName}, \{\}, \{\}, CT_0 \cup ET_0, 0)$$

where BasExName can be found in section 6.5 on page 57. CT_0 and ET_0 are defined below in figure 27. Setting n of s_0 to 0 is insignificant; any natural number would do. The reason for this arbitrariness is rule 299, which allows an arbitrary choice of question mark interpretation.

$$\begin{array}{lcl} CT_0 & = & \{\ (\texttt{true}, \texttt{bool}), (\texttt{false}, \texttt{bool}), (\texttt{nil}, \forall\texttt{'a.'a list}), \\ & & \quad (\texttt{::}, \forall\alpha.\{\texttt{1} \mapsto \texttt{'a,2} \mapsto \texttt{'a list}\} \to \texttt{'a list})\ \} \\ ET_0 & = & \{(excon, \texttt{exn}) \mid excon \in \text{BasExName} \setminus \{\texttt{NoCode}, \texttt{Abuse}\}\} \end{array}$$

Figure 27: Initial Value Templates

The set ET_0 does not contain the corresponding pairs for `NoCode` and `Abuse`, because they are not ordinary exceptions. We use the exception mechanism to propagate certain information, but we do not want quantified variables of type `exn` to range over these special exceptions.

C.2 The Initial Context for Derived Forms

The initial context for the derived forms C^0_{DER} has the form

$$C^0_{\text{DER}} \;=\; ((SE_{\text{DER}}, TE_{\text{DER}}, StE^0), VE^0_{\text{DER}}, F_{\text{DER}}, SE_{\text{DER}})$$

where all components except VE^0_{DER} and StE^0 are the empty map $\{\}$. VE^0_{DER} is defined in figure 28, StE^0 in figure 29. The set BasVal is defined in section 6.4, page 57. PredFun is the set containing the identifiers

```
o   @   ^   map   rev   not
```

The initial status environment StE^0 assigns identifier status to all predefined identifiers. Although the status `v` is the default status for identifiers in the semantics for derived forms, we still have to include value variables in StE^0, because the rules in the semantics for derived forms that require the generation of fresh variables take the condition $id \notin \text{Dom}\, StE$ as an indication of freshness.

id	$\mapsto$	(n,*fix*)	*condition*
o	$\mapsto$	(3,`l`)	
id	$\mapsto$	(4,`l`)	$id \in \{$`=`, `<>`, `<`, `>`, `<=`, `>=`$\}$
`@`	$\mapsto$	(5,`l`)	
`::`	$\mapsto$	(5,`r`)	
id	$\mapsto$	(6,`l`)	$id \in \{$`+`, `-`, `^`$\}$
id	$\mapsto$	(7,`l`)	$id \in \{$`div`, `mod`, `/`, `*`$\}$

Figure 28: Initial Infix Status VE^0_{DER}

id	$\mapsto$	*st*	*condition*
id	$\mapsto$	`e`	$id \in \mathrm{BasExName} \setminus \{$`NoCode`$\}$
id	$\mapsto$	`c`	$id \in \{$`true`, `false`, `::`, `nil`$\}$
id	$\mapsto$	`v`	$id \in \mathrm{BasVal} \cup \mathrm{PredFun}$

Figure 29: Initial Status Environment StE^0

C.3 The Initial Verification Basis

The initial verification basis B_0 has the form

$$B_0 \;=\; ((M_0, T_0), F_0, G_0, E_0)$$

where

- $M_0 \;=\; \emptyset$
- $T_0 \;=\; \{\texttt{bool}, \texttt{int}, \texttt{real}, \texttt{string}, \texttt{list}, \texttt{exn}\}$
- $F_0 \;=\; \{\}$
- $G_0 \;=\; \{\}$
- $E_0 \;=\; (SE_0, TE_0, VE_0)$
- $SE_0 = \{\}$

All type names in T_0 have arity 0, except `list` which has arity 1. All type names in T_0 except `exn` admit equality.

The initial type environment TE_0 is shown in figure 30. The initial variable environment VE_0 consists of two parts, $VE_0 = VE_0' \cup VE_0''$ with $\mathrm{Dom}\, VE_0' = \mathrm{BasVal} \cup \{\texttt{true}, \texttt{false}, \texttt{::}, \texttt{nil}\} \cup (\mathrm{BasExName} \setminus \{\texttt{NoCode}, \texttt{Abuse}\})$ and $\mathrm{Dom}\, VE_0'' = \mathrm{PredFun}$. All identifiers *id* in the domain of VE_0' are mapped to typed values *tv*. The type schemes Stat *tv* of the predefined identifiers are given in figure 31; the basic exceptions (not listed there) have all type `exn`.

tycon	$\mapsto$	$\{\ \theta,$	$\{con_1 \mapsto \sigma_1, \ldots, con_n \mapsto \sigma_n\}\ \}$ $(n \geq 0)$
`unit`	$\mapsto$	$\{\ \Lambda().\{\},$	$\{\}\ \}$
`bool`	$\mapsto$	$\{\ \texttt{bool},$	$\{\texttt{true} \mapsto \texttt{bool},\ \texttt{false} \mapsto \texttt{bool}\}\ \}$
`int`	$\mapsto$	$\{\ \texttt{int},$	$\{\}\ \}$
`real`	$\mapsto$	$\{\ \texttt{real},$	$\{\}\ \}$
`string`	$\mapsto$	$\{\ \texttt{string},$	$\{\}\ \}$
`list`	$\mapsto$	$\{\ \texttt{list},$	$\{\texttt{nil} \mapsto \forall\texttt{'a}\ .\ \texttt{'a list},$
			$\texttt{::} \mapsto \forall\texttt{'a}\ .\{1 \mapsto \texttt{'a}, 2 \mapsto \texttt{'a list}\} \to \texttt{'a list}\}\ \}$
`exn`	$\mapsto$	$\{\ \texttt{exn},$	$\{\}\ \}$

Figure 30: Static TE_0

NONFIX			INFIX		
var	$\mapsto$	σ	*var*	$\mapsto$	σ
map	$\mapsto$	$\forall$'a 'b. ('a $\to$ 'b) $\to$ 'a list $\to$ 'b list	Precedence 7 :		
rev	$\mapsto$	$\forall$'a. 'a list $\to$ 'a list	/	$\mapsto$	real $*$ real $\to$ real
not	$\mapsto$	bool $\to$ bool	div	$\mapsto$	int $*$ int $\to$ int
~	$\mapsto$	$\forall$num. num $\to$ num	mod	$\mapsto$	int $*$ int $\to$ int
abs	$\mapsto$	$\forall$num. num $\to$ num	*	$\mapsto$	$\forall$num. num $*$ num $\to$ num
floor	$\mapsto$	real $\to$ int	Precedence 6 :		
real	$\mapsto$	int $\to$ real	+	$\mapsto$	$\forall$num. num $*$ num $\to$ num
sqrt	$\mapsto$	real $\to$ real	-	$\mapsto$	$\forall$num. num $*$ num $\to$ num
sin	$\mapsto$	real $\to$ real	^	$\mapsto$	string $*$ string $\to$ string
cos	$\mapsto$	real $\to$ real	Precedence 5 :		
arctan	$\mapsto$	real $\to$ real	::	$\mapsto$	$\forall$'a.'a $*$ 'a list $\to$ 'a list
exp	$\mapsto$	real $\to$ real	@	$\mapsto$	$\forall$'a. 'a list $*$ 'a list $\to$ 'a list
ln	$\mapsto$	real $\to$ real	Precedence 4 :		
size	$\mapsto$	string $\to$ int	=	$\mapsto$	$\forall$''a. ''a $*$ ''a $\to$ bool
chr	$\mapsto$	int $\to$ string	<>	$\mapsto$	$\forall$''a. ''a $*$ ''a $\to$ bool
ord	$\mapsto$	string $\to$ int	<	$\mapsto$	$\forall$num. num $*$ num $\to$ bool
explode	$\mapsto$	string $\to$ string list	>	$\mapsto$	$\forall$num. num $*$ num $\to$ bool
implode	$\mapsto$	string list $\to$ string	<=	$\mapsto$	$\forall$num. num $*$ num $\to$ bool
true	$\mapsto$	bool	>=	$\mapsto$	$\forall$num. num $*$ num $\to$ bool
false	$\mapsto$	bool	Precedence 3 :		
nil	$\mapsto$	$\forall$'a. 'a list	o	$\mapsto$	$\forall$'a 'b 'c. ('b $\to$ 'c) $*$ ('a $\to$ 'b) $\to$ ('a $\to$ 'c)

Notes:

- In type schemes we have taken the liberty of writing $ty_1 * ty_2$ in place of $\{1 \mapsto ty_1, 2 \mapsto ty_2\}$.
- Recall that the special type variable num can be replaced by either real or int, see Section 4.5.

Figure 31: Static Variable Environment Stat VE_0 (without exceptions)

C.4 Predefined Functions

Note that $\mathrm{Dyn}\, VE'_0$ is the identity function; this is because we have chosen to denote basic values by the names of variables to which they are initially bound. We assume that we can always distinguish basic values from constructors — therefore, the bare identifiers are not values, but these identifiers attributed with their status.

The semantics of these basic values (most of which are functions) lies principally in their behaviour under APPLY, which we describe below.

VE''_0 contains initial variable bindings which, unlike BasVal, are definable in ML; it is obtained as follows: the following top declaration *topdec* is verificated with the program semantics (page 116) in the initial state and basis as $s_0, B'_0, C^0_{\mathrm{DER}} \vdash \mathit{topdec} \Rightarrow \mathcal{B}, C'_{\mathrm{DER}}$ where B'_0 is the same as B_0 but without VE''_0. $\mathcal{B} = \{(s, B)\}$ is a singleton set with $s = s_0$ and we get VE''_0 by removing all bindings in VE of E of B'_0 from VE of E of B.

```
fun (F o G)x = F(G x)

fun nil @ M = M
  | (x::L) @ M = x::(L @ M)

fun s ^ s' = implode((explode s) @ (explode s'))

fun map F nil = nil
  | map F (x::L) = (F x)::(map F L)

fun rev nil = nil
  | rev (x::L) = (rev L) @ [x]

fun not true = false
  | not false = true
```

We now describe the effect of APPLY upon each value $b \in$ BasVal. For special values, we shall normally use i, r, n, s to range over integers, reals, numbers (integer or real), strings respectively. We also take the liberty of abbreviating "APPLY(`abs`, r)" to "`abs`(r)", "APPLY(`mod`, $\{1 \mapsto i, 2 \mapsto d\}$)" to "$i$ `mod` d", etc.
.

- `~`(n) returns the negation of n, or the packet [`Neg`] if the result is out of range.
- `abs`(n) returns the absolute value of n, or the packet [`Abs`] if the result is out of range.
- `floor`(r) returns the largest integer i not greater than r; it returns the packet [`Floor`] if i is out of range.
- `real`(i) returns the real value equal to i.
- `sqrt`(r) returns the square root of r, or the packet [`Sqrt`] if r is negative.

- `sin`(r) , `cos`(r) return the result of the appropriate trigonometric functions.
- `arctan`(r) returns the result of the appropriate trigonometric function in the range $\pm\pi/2$.
- `exp`(r) , `ln`(r) return respectively the exponential and the natural logarithm of r, or an exception packet [`Exp`] or [`Ln`] if the result is out of range.
- `size`(s) returns the number of characters in s.
- `chr`(i) returns the character numbered i (see Section 2.2) if i is in the interval $[0, 255]$, and the packet [`Chr`] otherwise.
- `ord`(s) returns the number of the first character in s (an integer in the interval $[0, 255]$, see Section 2.2), or the packet [`Ord`] if s is empty.
- `explode`(s) returns the list of characters (as single-character strings) of which s consists.
- `implode`(L) returns the string formed by concatenating all members of the list L of strings.
- The arithmetic functions `/`,`*`,`+`,`-` all return the results of the usual arithmetic operations, or exception packets respectively [`Quot`], [`Prod`], [`Sum`], [`Diff`] if the result is undefined or out of range.
- i `mod` d , i `div` d return integers r, q (remainder, quotient) determined by the equation $d \times q + r = i$, where either $0 \leq r < d$ or $d < r \leq 0$. Thus the remainder has the same sign as the divisor d. The packet [`Mod`] or [`Div`] is returned if $d = 0$.
- The order relations `<`,`>`,`<=`,`>=` return boolean values in accord with their usual meanings.
- v_1 `=` v_2 returns `true` or `false` according as the values v_1 and v_2 are, or are not, identical. The type discipline (in particular, the fact that function types do not admit equality) ensures that equality is only ever applied to special values, nullary constructors, and values built out of such by record formation and constructor application.
- v_1 `<>` v_2 returns the opposite boolean value to v_1 `=` v_2.

References

[BHW94] M. Bidoit, R. Hennicker and M. Wirsing. Characterizing behavioural semantics and abstractor semantics. *Proc. 5th European Symposium on Programming*, Edinburgh. Springer LNCS 788, 105–119 (1994).

[Gol84] R. Goldblatt. *Topoi — The Categorical Analysis of Logic.* North-Holland (1984).

[Han93] J. Hannan. Extended natural semantics. *Journal of Functional Programming*, **3**(2), 123–152 (1993).

[Kah88] G. Kahn. Natural Semantics. In: K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North-Holland (1988).

[Kah93] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Report ECS-LFCS-93-257, Univ. of Edinburgh (1993).

[Kah94] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML – Addenda. Univ. of Edinburgh (1994).

[KST94] S. Kahrs and D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer Workshops in Computing, 186–215 (1994).

[MacQ86] D. MacQueen. Modules for Standard ML. In: R. Harper, D. MacQueen and R. Milner. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press (1991).

[MTH90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[San90] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park, 1990. Springer Workshops in Computing, 99–130 (1991).

[ST85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 67–77 (1985).

[ST86] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, 364–389 (1986).

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST91] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).