

The Definition of Extended ML

Stefan Kahrs* Donald Sannella* Andrzej Tarlecki†

— Version 1.21 — August 28, 1997

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.

Defects in This Document

This document contains the following (known) defects:

1. Behavioural equivalence. Mentioned as a defect in the introduction.
2. The way the meta-language deals with higher-order rules is not sufficient, since we can never show non-derivability, there are always “funny” models in which additional sentences are true. We probably need some kind of inductive semantics here (Andrzej).
3. In the dependency diagram there is an arrow missing, expressing the dependency between verification semantics for modules and static semantics for modules.
4. There are now some redundant existential quantifiers in several rules since the translation of rules has been changed. Similarly, the new function Tnames could be put to good use in the static semantics, getting rid of a couple of universal quantifiers.
5. Where exactly do Stephen Gilmore’s example of VDM-like specifications go wrong? They should be somehow, but it’s not clear that they do.
6. The defects section is not up-to-date.

*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.

Changes from version 1.0

The page numbers and rule numbers refer to ECS-LFCS-94-300.

Page 5. The translation of rules has been changed, meta-variables occurring in the conclusion but not in the premise of a rule are now implicitly existentially quantified (rather than universally). This made a couple of paragraphs on pages 6 and 7 (special care for higher order rules) redundant as well as several existential quantifiers throughout the semantics.

Page 8. A new keyword `iff` has been introduced. There is also a corresponding new entry in the grammar for the full language and a new rule in the derived form semantics.

Page 11 and 17. Due to an error in a macro definition the figures for syntactic objects did not display the names of the object classes.

Page 17. The condition for guardedness of structure expressions has been further restricted.

Page 24. The simple traces have slightly changed. There is an according change in rule 20.

Page 26. A function `Tnames` has been introduced for selecting the type names defined in a type environment.

Page 30. There are new side-conditions for rules 11.1–11.3 imposing that explicit type variables are abstracted where they are scoped.

Page 31, added a comment about the uniqueness of U in rules 11.1–11.3.

Page 32, The side-condition for explicit type variables in rule 17 (from SML) has been re-inserted. It is now based on the trace rather than the resulting variable environment. Similar side-conditions have been added to rules 11.1 to 11.3.

Page 36, added a comment to rule 46.

Page 38, the structure of traces has been slightly changed. Traces of the form (C, t) have been added.

Page 40, the class `TyRea` is introduced. It is later used (page 83) for the definition of state in the verification semantics.

Page 45, a stronger freshness condition is imposed. The old was unsound (and already unsound in SML).

Pages 46f. The principality requirement has been moved from rule 57.1 to rule 61.2. The only noticeable consequence of this move seems to be that overloading in axioms inside signatures has to be resolved. There is also a change in the actual principality requirement: the type of an axiomatic expression is now required to be always `bool`, ruling out expressions of type $\forall\alpha.\alpha$ which would necessarily be non-terminating or exception-raising. A direct consequence of this change was the need of a slight modification of the corresponding two rules in the verification semantics for modules.

Page 47, rule 61.2. Instead of using C of B for the elaboration of the expression we now add the names of B . This is necessary for several technical reasons, e.g. to allow `let`-expressions over specified types and also to ensure the soundness of axiom verification.

Page 48ff. Specifications are now prohibited to overwrite other specifications. Also, local specifications are required to use other identifiers than global specifications.

Page 57, \sim had been forgotten amongst the basic values.

Page 64, the comment to rule 106 was obsolete.

Page 65, a *space* inserted in missing rule 114.

Page 86, a function `Fresh` is introduced to deal with the renaming of type names in the verification semantics. This required certain changes in rules 241 and 242.

Page 90, `APPLY` now operates on a dynamic value.

Page 91f. The meaning of existential quantification (the `true`-case for `exists` and the `false`-case for `forall`) has changed: we now require witnesses for all types rather than a witness for the most general type.

Page 94f. The equality has been completely rewritten, especially the comparison of values. It is not longer possible to use `==` as logical equivalence, it is reserved to compare SML values only. Values are now compared in the state they occur — this solves a local soundness problem in the old presentation of comparison, in which values were compared in a state with possibly too few exception names. Another problem that has been addressed is that $s^{\#\#}$ is now applied to the type τ from the guess.

Page 99. Exception bindings are not any longer needed to be deterministically chosen, because equality works differently now. Therefore, we had to drop the explanation *why* the choice is deterministic.

Page 104. The combination of interfaces is greatly simplified, thanks to the earlier mentioned new restrictions on specifications.

Page 107, the second m is now m' (not strictly necessary, but conceptually cleaner).

Page 118. The phrase classes `InfExp` and `InfPat` have been exempted from the disambiguation principles.

The state has a new, fifth component l in the verification semantics. This component is initially set to \top . Whenever a “free” structure expression $strex\bullet$ is verified, it is done so in a state where l is set to \perp . The difference between both can be seen in rules 312 and 313: the latter (which is new) uses the *implementing* structure expressions to get the models of interest.

Related to the above change is that functor closures have an additional component, so-called *ostrex* for optional structure expression. This is the body of the original functor declaration. The new component is ignored as long as $l = \top$, but for $l = \perp$ it is activated — see rule 301. In words, the functor bodies can be used to create arbitrary well-formed structures.

Page 123, *strdec* second alternative: removed `withtype` (garbage).

Contents

Defects in This Document	i
Changes from version 1.0	ii
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 Infix 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 Infix 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
Names of a Type Environment	27
4.6 Scope of Explicit Type Variables	27
4.7 Non-expansive Expressions	27
4.8 Closure	27
4.9 Type Structures and Type Environments	28
4.10 Inference Rules	28
4.11 Further Restrictions	38

5	Static Semantics for Modules	39
5.1	Semantic Objects	39
5.2	Consistency	40
5.3	Well-formedness	40
5.4	Cycle-freedom	41
5.5	Admissibility	41
5.6	Type Realisation	41
5.7	Realisation	42
	Traces	42
	Stripping Axioms	42
5.8	Type Explication	43
5.9	Signature Instantiation	43
5.10	Functor Signature Instantiation	43
5.11	Enrichment	43
5.12	Signature Matching	44
5.13	Principal Signatures	44
5.14	Inference Rules	46
6	Dynamic Semantics for the Core	56
6.1	Reduced Syntax	56
6.2	Simple Objects	56
6.3	Compound Objects	57
6.4	Basic Values	58
6.5	Basic Exceptions	58
6.6	Closures	59
	States and Flags	59
	Pure Values	60
6.7	Inference Rules	64
7	Dynamic Semantics for Modules	74
7.1	Reduced Syntax	74
7.2	Semantic Objects	74
7.3	Inference Rules	77
8	Verification Semantics for the Core	83
8.1	Semantic Objects	83
8.2	Question Mark Interpretation	83
8.3	State	85
8.4	Value Access	86
8.5	Type Interpretation	86
8.6	Fresh Type Names	87
8.7	Projections to Dynamic and Static Semantics	87
8.8	Relationship to Dynamic Semantics	88
8.9	Sentences of Static and Dynamic Semantics	89

8.10 Inference Rules	89
9 Verification Semantics for Modules	103
9.1 Compound Objects	103
9.2 Generalised Axioms	104
9.3 Combining Interfaces	105
9.4 Extracting Objects of the Static Semantics	105
9.5 Sets	106
9.6 Inference Rules	106
10 Programs	116
A Appendix: Full Grammar	118
A.1 Syntactic Restrictions	125
B Appendix: Derived Forms	126
B.1 Semantic Objects	126
B.2 Identifiers	127
B.3 Enrichment	127
B.4 Inference Rules	127
C Appendix: The Initial Basis	144
C.1 The Initial State	144
C.2 The Initial Context for Derived Forms	144
C.3 The Initial Verification Basis	146
C.4 Predefined Functions	148
References	150

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 abstractions¹ 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:

¹The 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, Stephen Gilmore, Robert Harley, Martin Hofmann, Marcin Jurdzinski, Ed Kazmierczak, Mikolaj Konarski, Robin Milner, Claudio Russo, Aleksy Schubert, Mads Tofte and Judith Underwood 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

²A more obvious term is “verify”, but this carries connotations we would like to avoid.

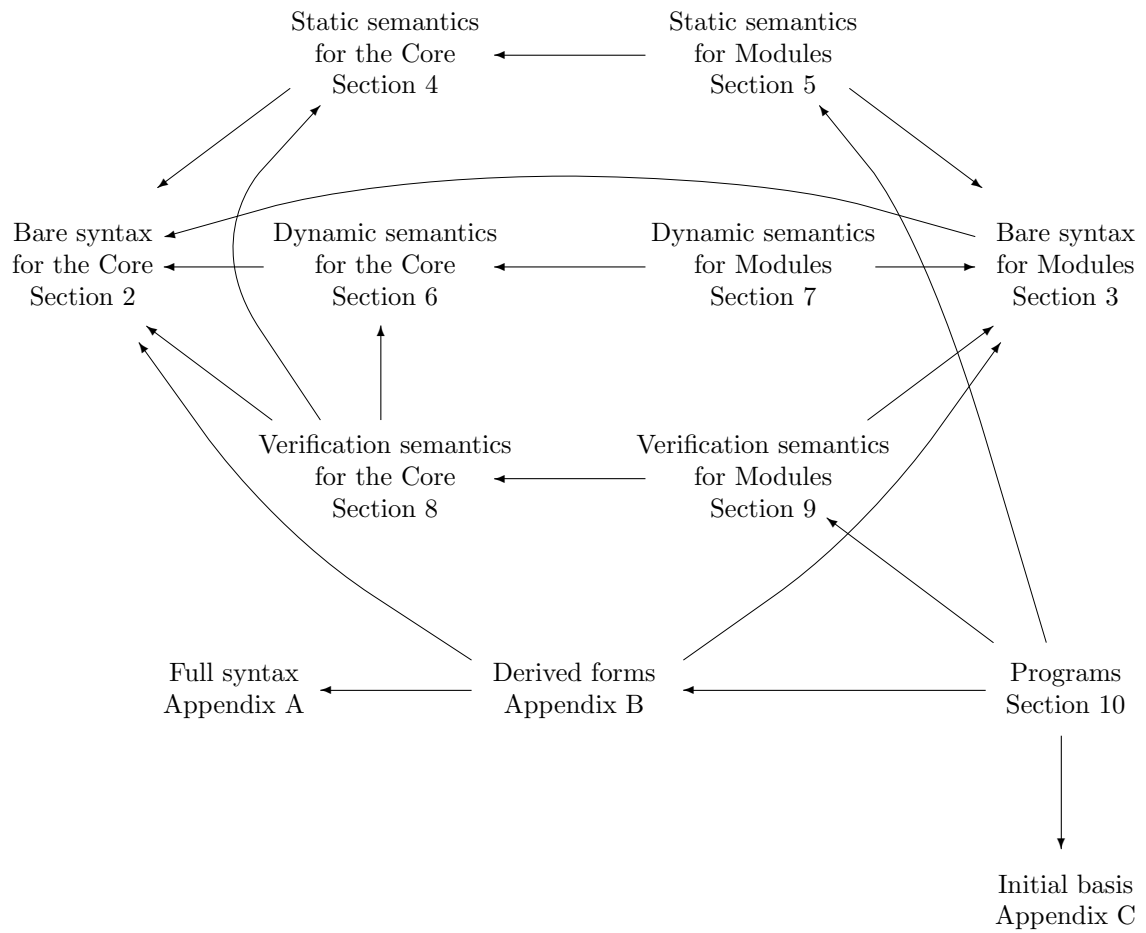


Figure 1: Parts of the definition of Extended ML

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 224–235) 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

$$\frac{v_1 \quad \cdots \quad v_k}{\psi}$$

where the *conclusion* ψ is a *sentence* and each *premise* (or *hypothesis*) v_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 \implies as a guarding operator in first-order logic then the formula $(P(x) \implies Q(x, y)) \implies R(y)$ is shorthand for $\forall y. (\forall x. (P(x) \implies Q(x, y)) \implies R(y))$, i.e. the variable x is bound at the inner implication, because it is not unguarded in the context $_ \implies R(y)$.

Let us make this precise. If v is a rule or sentence and V a set of variables, then $\llbracket v \rrbracket_V$ 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 v . The set V in $\llbracket v \rrbracket_V$ serves here as the context of v ; it should be understood as the set of variables occurring unguarded outside v . Similarly, if t is a term and V a set of variables, then $\llbracket t \rrbracket_V$ 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 as follows:

$$\begin{aligned} \llbracket \frac{v_1 \dots v_n}{\psi} \rrbracket &= \forall x_1 \dots \forall x_k. (\Upsilon_1 \wedge \dots \wedge \Upsilon_n \implies \Psi) \\ \text{where } \left\{ \begin{array}{l} (V_i, \Upsilon_i) &= \llbracket v_i \rrbracket_W \\ (V', \Psi) &= \llbracket \psi \rrbracket_W \\ W &= V_1 \cup \dots \cup V_n \cup V' \cup V_0 \\ \{x_1, \dots, x_k\} &= W \setminus V_0 \end{array} \right. \end{aligned}$$

Here, V_0 are names for (the components of) the initial basis, see Appendix C. For rules not occurring on outermost level the following translation scheme

applies:

$$\begin{aligned} \llbracket \frac{v_1 \dots v_n}{\psi} \rrbracket_V &= (\emptyset, \forall x_1. \dots \forall x_k. \exists y_1. \dots \exists y_l. (\Upsilon_1 \wedge \dots \wedge \Upsilon_n \implies \Psi)) \\ \text{where } \left\{ \begin{array}{l} (V_i, \Upsilon_i) &= \llbracket v_i \rrbracket_W \\ (V', \Psi) &= \llbracket \psi \rrbracket_W \\ W &= V_1 \cup \dots \cup V_n \cup V' \cup V \\ \{x_1, \dots, x_k\} &= (V_1 \cup \dots \cup V_n) \setminus V \\ \{y_1, \dots, y_l\} &= V' \setminus (V_1 \cup \dots \cup V_n \cup V) \end{array} \right. \end{aligned}$$

The properties maintained by the definition of $\llbracket v \rrbracket_V$ are that for $\llbracket v \rrbracket_V = (V', \phi)$, the free variables of ϕ are contained in $V \cup V'$, and if $\llbracket v \rrbracket_W = (W', \phi')$ then $V' = W'$, i.e. the set of unguarded variables does not depend on the second argument.

The above definition uniquely determines $\llbracket \frac{v_1 \dots v_n}{\psi} \rrbracket_V$ 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 v_i , the set of unguarded variables — this does not depend on W . The symbol \implies 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 $\llbracket \psi \rrbracket_V = (V', \Psi)$, 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 $p(A_1, \dots, A_n)$ 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 $f(A_1, \dots, A_n)$. Each variable occurs unguarded in itself: $\llbracket x \rrbracket_V = (\{x\}, x)$, 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].

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

Applying the translation $\llbracket \cdot \rrbracket_{V_0}$ to this rule, we obtain the following formula:

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

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.

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:

$$\begin{aligned} \llbracket \{A \mid \psi\} \rrbracket_V &= (\emptyset, \{y \mid \exists x_1 \dots \exists x_k. (y = A' \wedge \psi')\}) \\ \text{where } \left\{ \begin{array}{l} (V', \psi') &= \llbracket \psi \rrbracket_W \\ (W', A') &= \llbracket A \rrbracket_W \\ W &= V' \cup W' \cup V \\ \{x_1, \dots, x_k\} &= W \setminus V \end{array} \right. \end{aligned}$$

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:

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

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  iff  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 (\sim). A *real constant* is an integer constant, possibly followed by a point (\cdot) 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 $3.32E5$ $3E\sim7$ \cdot . Non-examples: 23 $\cdot3$ $4.E5$ $1E2.0$ \cdot .

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 \backslash , and stands for a character sequence. The escape sequences are:

$\backslash n$	A single character interpreted by the system as end-of-line.
$\backslash t$	Tab.
\backslash^c	The control character c , where c may be any character with number 64–95. The number of \backslash^c is 64 less than the number of c .
$\backslash ddd$	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...\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:

$$\begin{aligned} \textit{longx} ::= & x && \textit{identifier} \\ & \textit{strid}_1 \dots \textit{strid}_n . x && \textit{qualified identifier} \ (n \geq 1) \end{aligned}$$

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 exclude³ 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 ..., 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 $var \in \text{Var}$ has the form id^V , being an identifier $id \in \text{Id}$ 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 $E(id^S)$ is $SE(id)$ and $E(id^V)$ 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 Infix 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.

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

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
Mrule	match rules
Dec	declarations
ValBind	value bindings
TypBind	type bindings
DatBind	datatype bindings
ConBind	constructor bindings
ExBind	exception bindings
AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns
Ty	type expressions
TyRow	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:

$$\begin{aligned}
 xseq & ::= x && \text{(singleton sequence)} \\
 & && \text{(empty sequence)} \\
 & (x_1, \dots, x_n) && \text{(sequence, } n \geq 1)
 \end{aligned}$$

Note that the “...” 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 precedence; this resolves ambiguity in parsing, as explained in Appendix A.
- 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 X^\bullet (over which x^\bullet ranges) as the same as X, except that phrases of class 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 *tybind*, 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.

<i>atexp</i>	::=	<i>scon</i>	special constant
		<i>longvar</i>	value variable
		<i>longcon</i>	value constructor
		<i>longexcon</i>	exception constructor
		{ <i><exprow></i> }	record
		let <i>dec</i> in <i>exp</i> end	local declaration
		(<i>exp</i>)	
		?	undefined value
<i>exprow</i>	::=	<i>lab = exp</i> < , <i>exprow</i> >	expression row
<i>exp</i>	::=	<i>atexp</i>	atomic
		<i>exp atexp</i>	application
		<i>exp</i> : <i>ty</i>	typed
		<i>exp</i> [•] == <i>exp</i> [•] ₂	comparison (R)
		exists <i>match</i> [•]	existential quantifier
		forall <i>match</i> [•]	universal quantifier
		<i>exp</i> [•] terminates	convergence predicate
		<i>exp</i> handle <i>match</i>	handle exception
		raise <i>exp</i>	raise exception
		fn <i>match</i>	function
<i>match</i>	::=	<i>mrule</i> < <i>match</i> >	
<i>mrule</i>	::=	<i>pat</i> => <i>exp</i>	

Figure 4: Grammar: Expressions and Matches

<i>dec</i>	<code>::= val <i>valbind</i></code>	value declaration
	<code>type <i>typbind</i></code>	type declaration
	<code>eqtype <i>typbind</i></code>	equality type declaration
	<code>datatype <i>datbind</i></code>	datatype declaration
	<code>abstype <i>datbind</i> with <i>dec</i> end</code>	abstype declaration
	<code>exception <i>exbind</i></code>	exception declaration
	<code>local <i>dec</i>₁ in <i>dec</i>₂ end</code>	local declaration
	<code>open <i>longstrid</i>₁ ... <i>longstrid</i>_{<i>n</i>}</code>	open declaration ($n \geq 1$)
	<code><i>dec</i>₁ <;> <i>dec</i>₂</code>	sequential declaration
		empty declaration
<i>valbind</i>	<code>::= <i>pat</i> = <i>exp</i> <and <i>valbind</i>></code>	
	<code>rec <i>valbind</i></code>	
<i>typbind</i>	<code>::= <i>tyvarseq tycon</i> = <i>ty</i> <and <i>typbind</i>></code>	type binding
	<code><i>tyvarseq tycon</i> = ? <and <i>typbind</i>></code>	question mark type binding
<i>datbind</i>	<code>::= <i>tyvarseq tycon</i> = <i>conbind</i></code>	
	<code><and <i>datbind</i>></code>	
<i>conbind</i>	<code>::= <i>con</i> <of <i>ty</i>> < <i>conbind</i>></code>	
<i>exbind</i>	<code>::= <i>excon</i> <of <i>ty</i>> <and <i>exbind</i>></code>	
	<code><i>excon</i> = <i>longexcon</i> <and <i>exbind</i>></code>	

Figure 5: Grammar: Declarations and Bindings

$atpat$	$::=$	$_$	wildcard
		$scon$	special constant
		var	variable
		$longcon$	constant
		$longexcon$	exception constant
		$\{ \langle patrow \rangle \}$	record
		(pat)	
$patrow$	$::=$	\dots	wildcard
		$lab = pat \langle , patrow \rangle$	pattern row
pat	$::=$	$atpat$	atomic
		$longcon atpat$	value construction
		$longexcon atpat$	exception construction
		$pat : ty$	typed
		$var \langle : ty \rangle \text{ as } pat$	layered
ty	$::=$	$tyvar$	type variable
		$\{ \langle tyrow \rangle \}$	record type
		$tyseq longtycon$	type construction
		$ty \rightarrow ty'$	function type (R)
		(ty)	
$tyrow$	$::=$	$lab : ty \langle , tyrow \rangle$	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 380, necessarily obeys this restriction.

⁴The 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 reserved words listed in Section 2.1, Extended ML reserves the following words, which are used in the grammar for Modules:

<code>axiom</code>	<code>functor</code>	<code>include</code>	<code>sharing</code>
<code>sig</code>	<code>signature</code>	<code>struct</code>	<code>structure</code>

They may not be used as identifiers.

3.2 Identifiers

The additional identifier classes for Modules are SigId (signature identifiers) and FunId (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 Infix 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 *strex* 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*.

StrExp	structure expressions
StrDec	structure-level declarations
Ax	axioms
AxExp	axiomatic expressions
StrBind	structure bindings
SglStrBind	single structure bindings
SigExp	signature expressions
PSigExp	(principal) signature expressions
SigDec	signature declarations
SigBind	signature bindings
Spec	specifications
ValDesc	value descriptions
TypDesc	type descriptions
DatDesc	datatype descriptions
ConDesc	constructor descriptions
ExDesc	exception descriptions
AxDesc	axiom descriptions
SpecExp	specification expressions
StrDesc	structure descriptions
SharEq	sharing equations
FunDec	functor declarations
FunBind	functor bindings
TopDec	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*, *strex* must be guarded. A structure expression is called *guarded* if it is of the form *long-strid, funid (strexp)* or *let strdec in strexp end*, provided (in the last two cases) that *strex* 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 $strex$ are to functor and signature identifiers, except that both $psigexp'$ and $strex$ 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 49) and 96 (page 54) 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 40) 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.

<i>strexp</i>	::= struct <i>strdec</i> end <i>longstrid</i> <i>funid</i> (<i>strexp</i>) let <i>strdec</i> in <i>strexp</i> end	generative structure identifier functor application local declaration
<i>strdec</i>	::= <i>dec</i> axiom <i>ax</i> structure <i>strbind</i> local <i>strdec</i> ₁ in <i>strdec</i> ₂ end <i>strdec</i> ₁ <;> <i>strdec</i> ₂	declaration axiom structure local sequential empty
<i>ax</i>	::= <i>axexp</i> < and <i>ax</i> >	axiom
<i>axexp</i>	::= <i>exp</i> [•]	axiomatic expression
<i>strbind</i>	::= <i>sglstrbind</i> < and <i>strbind</i> >	structure binding
<i>sglstrbind</i>	::= <i>strid</i> : <i>psigexp</i> = <i>strexp</i> <i>strid</i> : <i>psigexp</i> = ? <i>strid</i> = <i>strexp</i>	single structure binding undefined structure binding unguarded structure binding
<i>sigexp</i>	::= sig <i>spec</i> end <i>sigid</i>	generative signature identifier
<i>psigexp</i>	::= <i>sigexp</i>	principal signature
<i>sigdec</i>	::= signature <i>sigbind</i> <i>sigdec</i> ₁ <;> <i>sigdec</i> ₂	single sequential empty
<i>sigbind</i>	::= <i>sigid</i> = <i>psigexp</i> < and <i>sigbind</i> >	

Figure 8: Grammar: Structure and Signature Expressions

<i>spec</i>	<code>::= val valdesc type typdesc eqtype eqtypdesc datatype datdesc exception exdesc axiom axdesc structure strdesc sharing shareq local spec₁ in spec₂ end open longstrid₁ ... longstrid_n include sigid₁ ... sigid_n spec₁ <;> spec₂</code>	value type eqtype datatype exception axiom structure sharing local open ($n \geq 1$) include ($n \geq 1$) sequential empty
<i>valdesc</i>	<code>::= var : ty <and valdesc></code>	
<i>typdesc</i>	<code>::= tyvarseq tycon <and typdesc></code>	
<i>datdesc</i>	<code>::= tyvarseq tycon = condesc <and datdesc></code>	
<i>condesc</i>	<code>::= con <of ty> < condesc></code>	
<i>exdesc</i>	<code>::= excon <of ty> <and exdesc></code>	
<i>axdesc</i>	<code>::= specexp <and axdesc></code>	
<i>specexp</i>	<code>::= let strdec in axexp end</code>	
<i>strdesc</i>	<code>::= strid : sigexp <and strdesc></code>	
<i>shareq</i>	<code>::= longstrid₁ = ... = longstrid_n type longtycon₁ = ... = longtycon_n shareq₁ and shareq₂</code>	structure sharing ($n \geq 2$) type sharing ($n \geq 2$) multiple

Figure 9: Grammar: Specifications

$fundec$	$::=$	<code>functor funbind</code> <code>fundec₁ <;> fundec₂</code>	single sequence empty
$funbind$	$::=$	<code>funid (strid : psigexp) : psigexp'</code> <code>= strexp <and funbind></code> <code>funid (strid : psigexp) : psigexp'</code> <code>= ? <and funbind></code>	functor binding undefined functor binding
$topdec$	$::=$	<code>strdec</code> <code>sigdec</code> <code>fundec</code>	structure-level declaration signature declaration 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 <i>tyvar</i>	\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 $\alpha \in \text{TyVar}$ 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 $t \in \text{TyName}$ 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 $\text{type}(scon)$ which is either `int`, `real` or `string` as indicated by Section 2.2.

4.2 Compound Objects

When A and B are sets $\text{Fin } A$ denotes the set of finite subsets of A , and $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B . The domain and range of a finite map, f , are denoted $\text{Dom } f$ and $\text{Ran } f$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \dots, a_k \mapsto b_k\}$, $k \geq 0$; 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 $|$ 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 $\text{Dom } f \cup \text{Dom } g$ and values

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

For any semantic object class A , we define $\text{Tree}(A)$ to be the finite binary trees of elements taken from A , i.e. $\text{Tree}(A)$ is the smallest solution of the domain equation $\text{Tree}(A) = \{\epsilon\} \uplus A \uplus \{x \cdot y \mid x, y \in \text{Tree}(A)\}$. 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 , $\text{tynames } A$ and $\text{tyvars } A$ 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 $C(\text{tycon})$ means $(\text{TE of } C)\text{tycon}$.

A particular case needs mention: $C(\text{con})$ is taken to stand for $(\text{VE of } C)\text{con}$. 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 $\text{longcon} = \text{strid}_1 \dots \text{strid}_k \cdot \text{con}$ then $E(\text{longcon})$ means

$$(\text{VE of } (\text{SE of } \dots (\text{SE of } (\text{SE of } E)\text{strid}_1)\text{strid}_2 \dots)\text{strid}_k)\text{con}.$$

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

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;
- $\{lab_1 \mapsto \tau_1, \dots, lab_n \mapsto \tau_n\}$, 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, $\Lambda\alpha.\alpha \rightarrow \alpha$ and $\Lambda\beta.\beta \rightarrow \beta$ 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 $\text{TyName} \subseteq \text{TypeFcn}$. $\theta = \Lambda\alpha^{(k)}. \tau$ 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 $\vartheta \in \text{TyVar} \xrightarrow{\text{fin}} \text{Type}$, where $\vartheta(\alpha)$ admits equality if α does, and where $\vartheta(\text{num})$ 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 $\tau\{\vartheta(\alpha)/\alpha ; \alpha \in \text{Dom } \vartheta\}$.

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 $\alpha^{(k)} \cap \text{tyvars}(\tau)$ 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 $\forall\alpha. \alpha \rightarrow \alpha$ and $\forall\beta. \beta \rightarrow \beta$ 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. The generalisation of traces \succ is defined as follows:

$$\begin{array}{llll}
\gamma & \succ & \gamma & \iff \gamma \in \text{SimTrace} \\
\gamma_1 \cdot \gamma_2 & \succ & \gamma'_1 \cdot \gamma'_2 & \iff \gamma_1 \succ \gamma'_1 \wedge \gamma_2 \succ \gamma'_2 \\
\forall\alpha^{(k)}. \gamma & \succ & \vartheta(\gamma) & \iff \text{Dom } \vartheta = \alpha^{(k)} \\
\gamma_1 & \succ & \gamma_3 & \iff \text{there exists } \gamma_2 \text{ such that } \gamma_1 \succ \gamma_2 \wedge \gamma_2 \succ \gamma_3 \\
\gamma_1 & \succ & \gamma_2 & \iff \text{for all } \gamma_3, \gamma_2 \succ \gamma_3 \implies \gamma_1 \succ \gamma_3
\end{array}$$

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

Names of a Type Environment

We define an auxiliary function $Tnames : TyEnv \rightarrow TyNameSet$ as follows:

$$Tnames(TE) = \{t \mid (t, CE) \in \text{Ran } TE\}$$

The purpose of this function is to select all type names defined in a type environment.

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

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

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

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

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 $CE(\text{con})$ of the form $\forall \alpha^{(k)}. (\tau \rightarrow \alpha^{(k)} t)$, the type function $\Lambda \alpha^{(k)}. \tau$ also admits equality. (This ensures that the 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

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

where no CE_i is the empty map, and for any E we define $\text{Abs}_C(TE, E)$ to be the environment obtained from E and TE as follows. First, let $\text{Abs}_C(TE)$ be the type environment $\{\text{tycon}_i \mapsto (t_i, \{\}) ; 1 \leq i \leq k\}$ in which all constructor environments CE_i have been replaced by the empty map. Let t'_1, \dots, t'_k be new distinct type names (“new” means that t'_1, \dots, t'_k are not in T of C), none of which admit equality. Then $\text{Abs}_C(TE, E)$ is the result of simultaneously substituting t'_i for t_i , $1 \leq i \leq k$, throughout $\text{Abs}_C(TE) + E$. (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 $\text{Supp}(\varphi_{\text{Ty}}) \subseteq \{t'_1, \dots, t'_k\}$ and $\varphi_{\text{Ty}}(t'_i) = t_i$, $1 \leq i \leq k$; we write $\text{Abs}_C(TE, E) =_{\varphi_{\text{Ty}}} E'$ to say that $\text{Abs}_C(TE, E) = E'$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{C \vdash \text{exp} \Rightarrow \tau, U, \gamma \quad \langle C \vdash \text{exprow} \Rightarrow \varrho, U', \gamma' \rangle}{C \vdash \text{lab} = \text{exp} \langle , \text{exprow} \rangle \Rightarrow \{\text{lab} \mapsto \tau\} \langle + \varrho \rangle, U \langle \cup U' \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (8)$$

Expressions

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

$$\frac{C \vdash \text{atexp} \Rightarrow \tau, U, \gamma}{C \vdash \text{atexp} \Rightarrow \tau, U, \gamma} \quad (9)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau' \rightarrow \tau, U, \gamma \quad C \vdash \text{atexp} \Rightarrow \tau', U', \gamma'}{C \vdash \text{exp atexp} \Rightarrow \tau, U \cup U', \gamma \cdot \gamma'} \quad (10)$$

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

$$\frac{C + U \vdash \text{exp}_1^\bullet \Rightarrow \tau, U_1, \gamma_1 \quad C + U \vdash \text{exp}_2^\bullet \Rightarrow \tau, U_2, \gamma_2 \quad U = U_1 \cup U_2 \quad \gamma' = \text{Clos}_C((C, \tau) \cdot \gamma_1 \cdot \gamma_2) \quad \text{tyvars } \gamma' \cap U \subseteq U \text{ of } C}{C \vdash \text{exp}_1^\bullet == \text{exp}_2^\bullet \Rightarrow \text{bool}, \emptyset, \gamma'} \quad (11.1)$$

$$\frac{C + U \vdash \text{match}^\bullet \Rightarrow \tau \rightarrow \text{bool}, U, \gamma \quad \gamma' = \text{Clos}_C((C, \tau) \cdot \gamma) \quad U \cap \text{tyvars } \gamma' \subseteq U \text{ of } C}{C \vdash \text{exists match}^\bullet \Rightarrow \text{bool}, \emptyset, \gamma'} \quad (11.2)$$

$$\frac{C + U \vdash \text{match}^\bullet \Rightarrow \tau \rightarrow \text{bool}, U, \gamma \quad \gamma' = \text{Clos}_C((C, \tau) \cdot \gamma) \quad U \cap \text{tyvars } \gamma' \subseteq U \text{ of } C}{C \vdash \text{forall match}^\bullet \Rightarrow \text{bool}, \emptyset, \gamma'} \quad (11.3)$$

$$\frac{C \vdash \text{exp}^\bullet \Rightarrow \tau, U, \gamma}{C \vdash \text{exp}^\bullet \text{ terminates} \Rightarrow \text{bool}, U, \epsilon} \quad (11.4)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau, U, \gamma \quad C \vdash \text{match} \Rightarrow \text{exn} \rightarrow \tau, U', \gamma'}{C \vdash \text{exp handle match} \Rightarrow \tau, U \cup U', \gamma \cdot \gamma'} \quad (12)$$

$$\frac{C \vdash \text{exp} \Rightarrow \text{exn}, U, \gamma}{C \vdash \text{raise exp} \Rightarrow \tau, U, \tau \cdot \gamma} \quad (13)$$

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

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 $(\text{fn } x \Rightarrow x) : 'a \rightarrow 'b$. 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 occurring 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.
- Superficially the set of unguarded variables is defined to be an arbitrary fixpoint, because U occurs left from “ \vdash ” and right from “ \Rightarrow ”. However, this fixpoint is unique: the unguarded variables of a syntactic object are independent from its context.
- 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

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

$$\frac{C \vdash \text{mrule} \Rightarrow \tau, U, \gamma \quad \langle C \vdash \text{match} \Rightarrow \tau, U', \gamma' \rangle}{C \vdash \text{mrule} \langle \mid \text{match} \rangle \Rightarrow \tau, U \langle \cup U' \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (15)$$

Match Rules

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

$$\frac{C \vdash \text{pat} \Rightarrow (VE, \tau), U, \gamma \quad C \oplus VE \vdash \text{exp} \Rightarrow \tau', U', \gamma'}{C \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \rightarrow \tau', U \cup U', \gamma \cdot \gamma'} \quad (16)$$

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

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

$$\frac{C + U \vdash \text{valbind} \Rightarrow VE, U, \gamma \quad VE' = \text{Clos}_C VE \quad \gamma' = \text{Clos}_C \gamma \quad U \cap \text{tyvars } \gamma' \subseteq U \text{ of } C}{C \vdash \text{val } \text{valbind} \Rightarrow VE' \text{ in Env}, \gamma'} \quad (17)$$

$$\frac{C \vdash \text{typbind} \Rightarrow TE, \gamma}{C \vdash \text{type } \text{typbind} \Rightarrow TE \text{ in Env}, TE \cdot \gamma} \quad (18)$$

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

$$\frac{C \oplus TE \vdash \text{datbind} \Rightarrow VE, TE, \gamma \quad \forall (t, CE) \in \text{Ran } TE, t \notin (T \text{ of } C) \quad E = (VE, TE) \text{ in Env} \quad TE \text{ maximises equality}}{C \vdash \text{datatype } \text{datbind} \Rightarrow E, E \cdot \gamma} \quad (19)$$

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

$$\frac{C \vdash \text{exbind} \Rightarrow VE, \gamma}{C \vdash \text{exception } \text{exbind} \Rightarrow VE \text{ in Env}, \gamma} \quad (21)$$

$$\frac{C \vdash \text{dec}_1 \Rightarrow E_1, \gamma_1 \quad C \oplus E_1 \vdash \text{dec}_2 \Rightarrow E_2, \gamma_2}{C \vdash \text{local } \text{dec}_1 \text{ in } \text{dec}_2 \text{ end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2} \quad (22)$$

$$\frac{C(\text{longstrid}_1) = (m_1, E_1) \quad \dots \quad C(\text{longstrid}_n) = (m_n, E_n)}{C \vdash \text{open } \text{longstrid}_1 \dots \text{longstrid}_n \Rightarrow E_1 + \dots + E_n, \epsilon} \quad (23)$$

$$\overline{C \vdash} \Rightarrow \{\} \text{ in Env}, \epsilon \quad (24)$$

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

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.

The side-condition on U ensures compatibility with SML, which requires that type variables are abstracted at their scoping construct.

(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 244 and 245.

(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

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

$$\frac{C \vdash pat \Rightarrow (VE, \tau), U, \gamma \quad C \vdash exp \Rightarrow \tau, U', \gamma' \quad \langle C \vdash valbind \Rightarrow VE', U'', \gamma'' \rangle}{C \vdash pat = exp \langle \text{and } valbind \rangle \Rightarrow VE \langle + VE' \rangle, U \cup U' \langle \cup U'' \rangle, \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle} \quad (26)$$

$$\frac{C + VE \vdash valbind \Rightarrow VE, U, \gamma}{C \vdash \text{rec } valbind \Rightarrow VE, U, \gamma} \quad (27)$$

Comments:

- (26) When the option is present we have $\text{Dom } VE \cap \text{Dom } VE' = \emptyset$ 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

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

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C \vdash \text{ty} \Rightarrow \tau \quad \langle C \vdash \text{typbind} \Rightarrow TE, \gamma \rangle}{C \vdash \text{tyvarseq } \text{tycon} = \text{ty} \langle \text{and } \text{typbind} \rangle \Rightarrow \{\text{tycon} \mapsto (\Lambda \alpha^{(k)}. \tau, \{\})\} \langle + TE \rangle, \epsilon \langle \cdot \gamma \rangle} \quad (28)$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad t \notin (T \text{ of } C), \text{ arity } t = k \quad \langle C + \{t\} \vdash \text{typbind} \Rightarrow TE, \gamma \rangle}{C \vdash \text{tyvarseq } \text{tycon} = ? \langle \text{and } \text{typbind} \rangle \Rightarrow \{\text{tycon} \mapsto (t, \{\})\} \langle + TE \rangle, (C, t) \langle \cdot \gamma \rangle} \quad (28.1)$$

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 $\text{tycon} \notin \text{Dom } TE$.
- (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 255; C and t are stored in the trace for the purposes of that replacement.

Data Type Bindings

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

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C, \alpha^{(k)} t \vdash \text{conbind} \Rightarrow CE, \gamma \quad \langle C \vdash \text{datbind} \Rightarrow VE, TE, \gamma' \quad \forall (t', CE) \in \text{Ran } TE, t \neq t' \rangle}{C \vdash \text{tyvarseq } \text{tycon} = \text{conbind} \langle \text{and } \text{datbind} \rangle \Rightarrow \text{Clos } CE \langle + VE \rangle, \{\text{tycon} \mapsto (t, \text{Clos } CE)\} \langle + TE \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (29)$$

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

Constructor Bindings

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

$$\frac{\text{con} = id^c \quad \langle C, \tau \vdash \text{conbind} \Rightarrow CE, \gamma \rangle}{C, \tau \vdash \text{con} \langle \mid \text{conbind} \rangle \Rightarrow \{id \mapsto \tau\} \langle + CE \rangle, \tau \langle \cdot \gamma \rangle} \quad (30)$$

$$\frac{\text{con} = id^c \quad C \vdash ty \Rightarrow \tau' \quad \tau'' = \tau' \rightarrow \tau \quad \langle C, \tau \vdash \text{conbind} \Rightarrow CE, \gamma \rangle}{C, \tau \vdash \text{con of } ty \langle \mid \text{conbind} \rangle \Rightarrow \{id \mapsto \tau''\} \langle + CE \rangle, \tau'' \langle \cdot \gamma \rangle} \quad (30.1)$$

Comments:

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

Exception Bindings

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

$$\frac{\text{excon} = id^e \quad \langle C \vdash \text{exbind} \Rightarrow VE, \gamma \rangle}{C \vdash \text{excon} \langle \text{and exbind} \rangle \Rightarrow \{id \mapsto \text{exn}\} \langle + VE \rangle, \epsilon \langle \cdot \gamma \rangle} \quad (31)$$

$$\frac{\text{excon} = id^e \quad C \vdash ty \Rightarrow \tau \quad \text{tyvars}(ty) = \emptyset \quad \langle C \vdash \text{exbind} \Rightarrow VE, \gamma \rangle}{C \vdash \text{excon of } ty \langle \text{and exbind} \rangle \Rightarrow \{id \mapsto \tau \rightarrow \text{exn}\} \langle + VE \rangle, \tau \langle \cdot \gamma \rangle} \quad (31.1)$$

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

Comments:

(31.1) Notice that ty must not contain any type variables. This is slightly stricter than to require $\text{tyvars } \tau = \emptyset$, 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 $C \vdash \text{exbind} \Rightarrow VE, \gamma$.

Atomic Patterns

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

$$\overline{C \vdash _ \Rightarrow (\{\}, \tau), \emptyset, \tau} \quad (33)$$

$$\overline{C \vdash \text{scon} \Rightarrow (\{\}, \text{type}(\text{scon})), \emptyset, \epsilon} \quad (34)$$

$$\frac{\text{var} = \text{id}^{\mathbf{V}}}{C \vdash \text{var} \Rightarrow (\{\text{id} \mapsto \tau\}, \tau), \emptyset, \tau} \quad (35)$$

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

$$\frac{C(\text{longexcon}) = \mathbf{exn}}{C \vdash \text{longexcon} \Rightarrow (\{\}, \mathbf{exn}), \emptyset, \epsilon} \quad (37)$$

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

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

Comments:

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

Pattern Rows

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

$$\overline{C \vdash \dots \Rightarrow (\{\}, \varrho), \emptyset, \varrho \text{ in Trace}} \quad (40)$$

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

Comments:

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

Patterns

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

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

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

$$\frac{C(longexcon) = \tau \rightarrow \mathbf{exn} \quad C \vdash atpat \Rightarrow (VE, \tau), U, \gamma}{C \vdash longexcon atpat \Rightarrow (VE, \mathbf{exn}), U, \gamma} \quad (44)$$

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

$$\frac{C \vdash var \Rightarrow (VE, \tau), U, \gamma \quad C \vdash pat \Rightarrow (VE', \tau), U', \gamma' \quad \langle C \vdash ty \Rightarrow \tau \quad U'' = \text{tyvars } ty \rangle}{C \vdash var \langle : ty \rangle \text{ as } pat \Rightarrow (VE + VE', \tau), U \cup U' \langle \cup U'' \rangle, \gamma \cdot \gamma'} \quad (46)$$

Comments:

- (46) By the syntactic restrictions, $\text{Dom } VE \cap \text{Dom } VE' = \emptyset$. In the first premise, *var* is viewed as an atomic pattern. Note that *U* will always be the empty set.

Type Expressions

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

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \quad (47)$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \{ \langle tyrow \rangle \} \Rightarrow \{ \} \langle + \varrho \rangle \text{ in Type}} \quad (48)$$

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

$$\frac{C \vdash ty \Rightarrow \tau \quad C \vdash ty' \Rightarrow \tau'}{C \vdash ty \rightarrow ty' \Rightarrow \tau \rightarrow \tau'} \quad (50)$$

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

Type-expression Rows

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

$$\frac{C \vdash ty \Rightarrow \tau \quad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab : ty \langle , tyrow \rangle \Rightarrow \{ lab \mapsto \tau \} \langle + \varrho \rangle} \quad (52)$$

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

4.11 Further Restrictions

In a match of the form $pat_1 \Rightarrow exp_1 \mid \dots \mid pat_n \Rightarrow exp_n$ the pattern sequence pat_1, \dots, 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 atpat1 ... atpatn (: ty) = 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.

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

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, $\text{strnames}(A)$ and $\text{names}(A)$ 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 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 $B + (\text{names } SE, SE)$.

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 $S_1(\text{longstrid})$ and $S_2(\text{longstrid})$ exist, then

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

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

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

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 $N \subseteq \text{names } S$, and also, whenever (m, E) is a substructure of S and $m \notin N$, then $N \cap (\text{names } E) = \emptyset$, and whenever (t, CE) is a type structure of S and $t \notin N$, then $N \cap (\text{names } CE) = \emptyset$. 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

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

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 $A \vdash \textit{phrase} \Rightarrow A'$ inferred by the rules given in Section 5.14, the assembly $\{A, A'\}$ is admissible.
2. In the special case of a sentence $B \vdash \textit{sigexp} \Rightarrow S, \gamma$, 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 $\varphi_{\text{Ty}} : \text{TyName} \rightarrow \text{TypeFcn}$ 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 $\varphi_{\text{Ty}}(t) \neq t$.

The semantic class of type realisations is called TyRea .

5.7 Realisation

A *realisation* is a function φ of names, partitioned into a type realisation $\varphi_{\text{Ty}} : \text{TyName} \rightarrow \text{TypeFcn}$ and a function $\varphi_{\text{Str}} : \text{StrName} \rightarrow \text{StrName}$. The *support* $\text{Supp } \varphi$ of a realisation φ is the set of names n for which $\varphi(n) \neq n$. The *yield* $\text{Yield } \varphi$ of a realisation φ is the set of names which occur in some $\varphi(n)$ for which $n \in \text{Supp } \varphi$.

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) ,

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

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 extending the relation \succ_{COR} on *Core* traces in the following way:

$$\begin{array}{llll} \gamma & \succ & \gamma & \iff \gamma \in \text{SimTrace} \\ \gamma_1 \cdot \gamma_2 & \succ & \gamma'_1 \cdot \gamma'_2 & \iff \gamma_1 \succ \gamma'_1 \wedge \gamma_2 \succ \gamma'_2 \\ (N)\gamma & \succ & \varphi(\gamma) & \iff \text{Supp } \varphi \subseteq N \\ \gamma_1 & \succ & \gamma_3 & \iff \text{there exists } \gamma_2 \text{ such that } \gamma_1 \succ \gamma_2 \wedge \gamma_2 \succ \gamma_3 \\ \gamma_1 & \succ & \gamma_2 & \iff \text{for all } \gamma_3, \gamma_2 \succ \gamma_3 \implies \gamma_1 \succ \gamma_3 \end{array}$$

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

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

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, γ) of *strip* is expected to be an elaboration result of *spec*. More precisely, *strip* is defined as follows:

$$\begin{aligned} \text{strip} & : (\text{SigExp} \times \text{Trace}) \rightarrow (\text{SigExp} \times \text{Trace}) \\ \text{strip}(\mathbf{sig} \text{ spec } \mathbf{end}, m \cdot \gamma) & = (\mathbf{sig} \text{ spec}' \mathbf{end}, m \cdot \gamma') \\ & \text{where } \text{strip}(\text{spec}, \gamma) = (\text{spec}', \gamma') \\ \text{strip}(\text{sigid}, \gamma) & = (\text{sigid}, \gamma) \end{aligned}$$

$$\begin{aligned}
& \text{strip} : (\text{StrDesc} \times \text{Trace}) \rightarrow (\text{StrDesc} \times \text{Trace}) \\
& \text{strip}(\text{strid} : \text{sigexp} \langle \text{and strdesc} \rangle, \gamma_1 \langle \cdot \gamma_2 \rangle) = \\
& \quad (\text{strid} : \text{sigexp}' \langle \text{and strdesc}' \rangle, \gamma_1' \langle \cdot \gamma_2' \rangle) \\
& \quad \text{where } \left\{ \begin{array}{l} \text{strip}(\text{sigexp}, \gamma_1) = (\text{sigexp}', \gamma_1') \\ \langle \text{strip}(\text{strdesc}, \gamma_2) = (\text{strdesc}', \gamma_2') \rangle \end{array} \right. \\
& \text{strip} : (\text{Spec} \times \text{Trace}) \rightarrow (\text{Spec} \times \text{Trace}) \\
& \text{strip}(\text{axiom axdesc}, \gamma) = (\quad, \epsilon) \\
& \text{strip}(\text{structure strdesc}, \gamma) = (\text{structure strdesc}', \gamma') \\
& \quad \text{where } \text{strip}(\text{strdesc}, \gamma) = (\text{strdesc}', \gamma') \\
& \text{strip}(\text{local spec}_1 \text{ in spec}_2 \text{ end}, \gamma_1 \cdot \gamma_2) = (\text{local spec}'_1 \text{ in spec}'_2 \text{ end}, \gamma_1' \cdot \gamma_2') \\
& \quad \text{where } \text{strip}(\text{spec}_i, \gamma_i) = (\text{spec}'_i, \gamma_i'), i \in \{1, 2\} \\
& \text{strip}(\text{spec}_1 \langle ; \rangle \text{spec}_2, \gamma_1 \cdot \gamma_2) = (\text{spec}'_1 \langle ; \rangle \text{spec}'_2, \gamma_1' \cdot \gamma_2') \\
& \quad \text{where } \text{strip}(\text{spec}_i, \gamma_i) = (\text{spec}'_i, \gamma_i'), i \in \{1, 2\} \\
& \text{strip}(\text{spec}, \gamma) = (\text{spec}, \gamma) \quad \text{otherwise}
\end{aligned}$$

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 $TE(\text{tycon}) = (t, CE)$ 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 $\text{Supp } \varphi \subseteq N_1$. 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 $N_2 \cap (\text{names } \Sigma_1) = \emptyset$. 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 $\text{Supp } \varphi \subseteq N_1$. 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. $\text{Dom } SE_1 \supseteq \text{Dom } SE_2$, and $\forall \text{strid} \in \text{Dom } SE_2. SE_1(\text{strid}) \succ SE_2(\text{strid})$
2. $\text{Dom } TE_1 \supseteq \text{Dom } TE_2$, and $\forall \text{tycon} \in \text{Dom } TE_2. TE_1(\text{tycon}) \succ TE_2(\text{tycon})$
3. $\text{Dom } VE_1 \supseteq \text{Dom } VE_2$, and $\forall \text{id} \in \text{Dom } VE_2. VE_1(\text{id}) \succ VE_2(\text{id})$

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 $\text{Supp } \varphi \subseteq N_1$ 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 $\text{Supp } \varphi \subseteq N_1$ and $\varphi(S_1) \prec S_2$ and $N_2 \cap \text{names } \Sigma_1 = \emptyset$.

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 $B \vdash \text{sigexp} \Rightarrow S, \gamma$, 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 $strid \in \text{Dom } E$, B contains a substructure (m, E') with m free and $strid \in \text{Dom } E'$
2. For every type constructor $tycon \in \text{Dom } E$, B contains a substructure (m, E') with m free and $tycon \in \text{Dom } E'$

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

⁵This 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

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

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

$$\boxed{B \vdash \textit{strex} \Rightarrow S, \gamma}$$

$$\frac{B \vdash \textit{strdec} \Rightarrow E, \gamma \quad m \notin (N \text{ of } B) \cup \text{names } E}{B \vdash \mathbf{struct} \textit{strdec} \mathbf{end} \Rightarrow (m, E), m \cdot \gamma} \quad (53)$$

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

$$\frac{B \vdash \textit{strex} \Rightarrow S, \gamma \quad B(\textit{funid})_{\geq \varphi}(S'', (N')S'), S \succ S'' \quad (N \text{ of } B \cup \text{names } S) \cap N' = \emptyset}{B \vdash \textit{funid} (\textit{strex}) \Rightarrow S', \varphi \cdot \gamma} \quad (55)$$

$$\frac{B \vdash \textit{strdec} \Rightarrow E, \gamma \quad B \oplus E \vdash \textit{strex} \Rightarrow S, \gamma'}{B \vdash \mathbf{let} \textit{strdec} \mathbf{in} \textit{strex} \mathbf{end} \Rightarrow S, \gamma \cdot \gamma'} \quad (56)$$

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 $(N \text{ of } B \cup \text{names } S) \cap N' = \emptyset$ can always be satisfied by renaming bound names in $(N')S'$ thus ensuring that the generated structures receive new names. The corresponding side condition in rule 55 of the SML semantics [MTH90] is weaker, but this weakness makes the rule unsound.

Let $B(\textit{funid}) = (N)(S_f, (N')S'_f)$. 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 \textit{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

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

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

$$\frac{B \vdash \text{ax} \Rightarrow \gamma}{B \vdash \text{axiom } \text{ax} \Rightarrow \{\} \text{ in Env}, \gamma} \quad (57.1)$$

$$\frac{B \vdash \text{strbind} \Rightarrow SE, \gamma}{B \vdash \text{structure } \text{strbind} \Rightarrow SE \text{ in Env}, \gamma} \quad (58)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1, \gamma_1 \quad B \oplus E_1 \vdash \text{strdec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \text{local } \text{strdec}_1 \text{ in } \text{strdec}_2 \text{ end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2} \quad (59)$$

$$\overline{B \vdash \quad \Rightarrow \{\} \text{ in Env}, \epsilon} \quad (60)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1, \gamma_1 \quad B \oplus E_1 \vdash \text{strdec}_2 \Rightarrow E_2, \gamma_2}{B \vdash \text{strdec}_1 \langle ; \rangle \text{strdec}_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2} \quad (61)$$

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 .

Axioms

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

$$\frac{B \vdash \text{axexp} \Rightarrow \gamma \quad \langle B \vdash \text{ax} \Rightarrow \gamma' \rangle}{B \vdash \text{axexp} \langle \text{and } \text{ax} \rangle \Rightarrow \gamma \langle \cdot \gamma' \rangle} \quad (61.1)$$

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

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

$$\frac{C \text{ of } B + \text{names } B \vdash exp^\bullet \Rightarrow \text{bool}, \emptyset, \gamma \quad \frac{C \text{ of } B + \text{names } B \vdash exp^\bullet \Rightarrow \tau, \emptyset, \gamma'}{\text{Clos } \gamma \succ \gamma' \wedge \tau = \text{bool}}}{B \vdash exp^\bullet \Rightarrow \text{Clos } \gamma} \quad (61.2)$$

Comment: This rule gives us the principal trace for an expression in an axiom or axiom description; principality is given by the second premise. Axiomatic expressions must be of type `bool` and are not allowed to contain unguarded explicit type variables.

Structure Bindings

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

$$\frac{B \vdash sglstrbind \Rightarrow SE, \gamma \quad \langle B + \text{names } SE \vdash strbind \Rightarrow SE', \gamma' \rangle}{B \vdash sglstrbind \langle \text{and } strbind \rangle \Rightarrow SE \langle +SE' \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (62)$$

Single Structure Bindings

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

$$\frac{B \vdash psigexp \Rightarrow (N)S, \gamma \quad B \vdash strexp \Rightarrow S', \gamma' \quad N \cap N \text{ of } B = \emptyset \quad (N)S \geq S'' \prec S'}{B \vdash strid : psigexp = strexp \Rightarrow \{strid \mapsto S\}, \gamma \cdot \gamma'} \quad (62.1)$$

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

$$\frac{B \vdash strexp \Rightarrow S, \gamma}{B \vdash strid = strexp \Rightarrow \{strid \mapsto S\}, \gamma} \quad (62.3)$$

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 $N \cap N \text{ of } B = \emptyset$ can always be satisfied by an appropriate α -conversion of $(N)S$.

Signature Expressions

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

$$\frac{B, E \vdash \text{spec} \Rightarrow E, \gamma}{B \vdash \text{sig spec end} \Rightarrow (m, E), m \cdot \gamma} \quad (63)$$

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

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.

The purpose of the recursive occurrence of E is to ensure that the bindings present in the final result of a signature expression are never overwritten during its elaboration.

- (64) The instance S of $B(\text{sigid})$ 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

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

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

Comment: $B \vdash \text{sigexp} \Rightarrow S, \gamma$ 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

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

$$\frac{B \vdash \text{sigbind} \Rightarrow G, \gamma}{B \vdash \text{signature sigbind} \Rightarrow G, \gamma} \quad (66)$$

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

$$\frac{B \vdash \text{sigdec}_1 \Rightarrow G_1, \gamma_1 \quad B + G_1 \vdash \text{sigdec}_2 \Rightarrow G_2, \gamma_2}{B \vdash \text{sigdec}_1 \langle ; \rangle \text{sigdec}_2 \Rightarrow G_1 + G_2, \gamma_1 \cdot \gamma_2} \quad (68)$$

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

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

$$\frac{B \vdash \text{psigexp} \Rightarrow \Sigma, \gamma \quad \langle B \vdash \text{sigbind} \Rightarrow G, \gamma' \rangle}{B \vdash \text{sigid} = \text{psigexp} \langle \text{and sigbind} \rangle \Rightarrow \{\text{sigid} \mapsto \Sigma\} \langle + G \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (69)$$

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

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

$$\frac{C \text{ of } B \vdash \text{valdesc} \Rightarrow VE}{B, E \vdash \text{val valdesc} \Rightarrow \text{Clos}VE \text{ in Env, Clos}VE \text{ in Trace}} \quad (70)$$

$$\frac{C \text{ of } B \vdash \text{typdesc} \Rightarrow TE}{B, E \vdash \text{type typdesc} \Rightarrow TE \text{ in Env, TE in Trace}} \quad (71)$$

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

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

$$\frac{C \text{ of } B \vdash \text{exdesc} \Rightarrow VE \quad E = (\{\}, \{\}, VE)}{B, E' \vdash \text{exception exdesc} \Rightarrow E, E \text{ in Trace}} \quad (74)$$

$$\frac{B \vdash \text{axdesc} \Rightarrow \gamma}{B, E \vdash \text{axiom axdesc} \Rightarrow \{\} \text{ in Env, } \gamma} \quad (74.1)$$

$$\frac{B \vdash \text{strdesc} \Rightarrow SE, \gamma}{B, E \vdash \text{structure strdesc} \Rightarrow SE \text{ in Env, } \gamma} \quad (75)$$

$$\frac{B \vdash \text{shareq} \Rightarrow \{\}}{B, E \vdash \text{sharing shareq} \Rightarrow \{\} \text{ in Env, } \epsilon} \quad (76)$$

$$\frac{\text{Dom } E_1 \cap \text{Dom } E = \emptyset \quad \text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset}{B, E \vdash \text{spec}_1 \Rightarrow E_1, \gamma_1 \quad B + E_1, E \vdash \text{spec}_2 \Rightarrow E_2, \gamma_2} \quad (77)$$

$$B, E \vdash \text{local spec}_1 \text{ in spec}_2 \text{ end} \Rightarrow E_2, \gamma_1 \cdot \gamma_2$$

$$\frac{\forall 1 \leq i < j \leq n. \text{Dom } E_i \cap \text{Dom } E_j = \emptyset}{B(\text{longstrid}_1) = (m_1, E_1) \quad \dots \quad B(\text{longstrid}_n) = (m_n, E_n)} \quad (78)$$

$$B, E \vdash \text{open longstrid}_1 \dots \text{longstrid}_n \Rightarrow E_1 + \dots + E_n, \epsilon$$

$$\frac{\forall 1 \leq i < j \leq n. \text{Dom } E_i \cap \text{Dom } E_j = \emptyset}{B(\text{sigid}_1)_{\geq \varphi_1}(m_1, E_1) \quad \dots \quad B(\text{sigid}_n)_{\geq \varphi_n}(m_n, E_n)} \quad (79)$$

$$B, E \vdash \text{include sigid}_1 \dots \text{sigid}_n \Rightarrow E_1 + \dots + E_n, \varphi_1 \cdot \dots \cdot \varphi_n$$

$$\overline{B, E \vdash \quad \Rightarrow \{ \} \text{ in Env, } \epsilon} \quad (80)$$

$$\frac{\text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset}{B, E \vdash \text{spec}_1 \Rightarrow E_1, \gamma_1 \quad B + E_1, E \vdash \text{spec}_2 \Rightarrow E_2, \gamma_2} \quad (81)$$

$$B, E \vdash \text{spec}_1 \langle ; \rangle \text{spec}_2 \Rightarrow E_1 + E_2, \gamma_1 \cdot \gamma_2$$

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.

(77)–(79),(81) The intersection of the domains is to be understood as a point-wise intersection, the name spaces of e.g. structure identifiers and type constructors do not interfere. The reason for the restriction is to simplify the verification of structures against signatures.

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

Value Descriptions

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

$$\frac{\text{var} = \text{id}^V \quad C \vdash \text{ty} \Rightarrow \tau \quad \langle C \vdash \text{valdesc} \Rightarrow VE \rangle}{C \vdash \text{var} : \text{ty} \langle \text{and valdesc} \rangle \Rightarrow \{ \text{id} \mapsto \tau \} \langle + VE \rangle} \quad (82)$$

Type Descriptions

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

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

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 \mathfrak{t} does not bind any value constructors.

Datatype Descriptions

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

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C, \alpha^{(k)}t \vdash \text{condesc} \Rightarrow CE \quad \langle C \vdash \text{datdesc} \Rightarrow VE, TE \rangle}{C \vdash \text{tyvarseq } \text{tycon} = \text{condesc} \langle \text{and } \text{datdesc} \rangle \Rightarrow \text{ClosCE} \langle + VE \rangle, \{\text{tycon} \mapsto (t, \text{ClosCE})\} \langle + TE \rangle} \quad (84)$$

Constructor Descriptions

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

$$\frac{\text{con} = id^C \quad \langle C, \tau \vdash \text{condesc} \Rightarrow CE \rangle}{C, \tau \vdash \text{con} \langle | \text{condesc} \rangle \Rightarrow \{id \mapsto \tau\} \langle + CE \rangle} \quad (85)$$

$$\frac{\text{con} = id^C \quad C \vdash \text{ty} \Rightarrow \tau' \quad \langle C, \tau \vdash \text{condesc} \Rightarrow CE \rangle}{C, \tau \vdash \text{con of } \text{ty} \langle | \text{condesc} \rangle \Rightarrow \{id \mapsto \tau' \rightarrow \tau\} \langle + CE \rangle} \quad (85.1)$$

Exception Descriptions

$$\boxed{C \vdash \text{exdesc} \Rightarrow VE}$$

$$\frac{\text{excon} = id^e \quad \langle C \vdash \text{exdesc} \Rightarrow VE \rangle}{C \vdash \text{excon} \langle \text{and } \text{exdesc} \rangle \Rightarrow \{id \mapsto \text{exn}\} \langle + VE \rangle} \quad (86)$$

$$\frac{C \vdash \text{ty} \Rightarrow \tau \quad \text{tyvars}(\text{ty}) = \emptyset \quad \text{excon} = id^e \quad \langle C \vdash \text{exdesc} \Rightarrow VE \rangle}{C \vdash \text{excon of } \text{ty} \langle \text{and } \text{exdesc} \rangle \Rightarrow \{id \mapsto \tau \rightarrow \text{exn}\} \langle + VE \rangle} \quad (86.1)$$

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 $\Lambda\alpha.\text{int}$; 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 $\text{tyvars}(\tau) = \emptyset$ would still allow successful elaborations with e.g. $\mathbf{t} \mapsto \Lambda\alpha.\text{int}$.

Axiom Descriptions

$$\boxed{B \vdash \text{axdesc} \Rightarrow \gamma}$$

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

$$\frac{B \vdash \text{specexp} \Rightarrow \gamma \quad \langle B \vdash \text{axdesc} \Rightarrow \gamma' \rangle}{B \vdash \text{specexp} \langle \text{and axdesc} \rangle \Rightarrow \gamma \langle \cdot \gamma' \rangle} \quad (86.2)$$

Specification Expressions

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

$$\frac{B \vdash \text{strdec} \Rightarrow E, \gamma \quad B \oplus E \vdash \text{axexp} \Rightarrow \gamma'}{B \vdash \text{let strdec in axexp end} \Rightarrow \gamma \cdot \gamma'} \quad (86.3)$$

Structure Descriptions

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

$$\frac{B \vdash \text{sigexp} \Rightarrow S, \gamma \quad \langle B \vdash \text{strdesc} \Rightarrow SE, \gamma' \rangle}{B \vdash \text{strid} : \text{sigexp} \langle \text{and strdesc} \rangle \Rightarrow \{\text{strid} \mapsto S\} \langle + SE \rangle, \gamma \langle \cdot \gamma' \rangle} \quad (87)$$

Sharing Equations

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

$$\frac{m \text{ of } B(\text{longstrid}_1) = \dots = m \text{ of } B(\text{longstrid}_n)}{B \vdash \text{longstrid}_1 = \dots = \text{longstrid}_n \Rightarrow \{\}} \quad (88)$$

$$\frac{\theta \text{ of } B(\text{longtycon}_1) = \dots = \theta \text{ of } B(\text{longtycon}_n)}{B \vdash \text{type longtycon}_1 = \dots = \text{longtycon}_n \Rightarrow \{\}} \quad (89)$$

$$\frac{B \vdash \text{shareq}_1 \Rightarrow \{\} \quad B \vdash \text{shareq}_2 \Rightarrow \{\}}{B \vdash \text{shareq}_1 \text{ and } \text{shareq}_2 \Rightarrow \{\}} \quad (90)$$

Comments:

(88) The premise is weaker than $B(\text{longstrid}_1) = \dots = B(\text{longstrid}_n)$. 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 $B(\text{longtycon}_1) = \dots = B(\text{longtycon}_n)$. 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.

$$\text{Deleted} \tag{91}$$

$$\text{Deleted} \tag{92}$$

$$\text{Deleted} \tag{93}$$

$$\text{Deleted} \tag{94}$$

$$\text{Deleted} \tag{95}$$

Functor Declarations

$$\boxed{B \vdash \text{fundec} \Rightarrow F, \gamma}$$

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

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

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

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

$$\boxed{B \vdash \text{funbind} \Rightarrow F, \gamma}$$

$$\frac{B \vdash \text{psigexp} \Rightarrow (N)S, \gamma_1 \quad N \cap N \text{ of } B = \emptyset \quad B' = B \oplus \{\text{strid} \mapsto S\} \\ B' \vdash \text{psigexp}' \Rightarrow \Sigma, \gamma_2 \quad B' \vdash \text{strex} \Rightarrow S', \gamma_3 \\ \Sigma \geq S'' \prec S' \quad \langle B \vdash \text{funbind} \Rightarrow F, \gamma_4 \rangle}{B \vdash \text{funid} (\text{strid} : \text{psigexp}) : \text{psigexp}' = \text{strex} \langle \text{and funbind} \rangle \Rightarrow \\ \{\text{funid} \mapsto (N)(S, \Sigma)\} \langle + F \rangle, \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \langle \cdot \gamma_4 \rangle} \quad (99)$$

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

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

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

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

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

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

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 `TyBind`, `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.

$en \in$	<code>ExName</code>	exception names
$b \in$	<code>BasVal</code>	basic values
$sv \in$	<code>SVal</code>	special values
$sp \in$	<code>Bit = {\perp, \top}</code>	flags
	<code>{FAIL}</code>	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 $\text{val}(scon)$. The values \perp and \top are only used as Boolean flags. If a semantic object x contains a `Bit` component, we write x_{\perp} to denote the same object, but with the `Bit` component set to \perp . On `Bit`, we define the operation \wedge as the greatest lower bound of the order $\perp \leq \top$; 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.

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

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 $\text{Str} = \text{StrName} \times \text{Env}$ 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 $\text{BasExName} \subset \text{ExName}$ 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 $\sim / * + -$ 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 = exp1 and y = exp2` 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`”, “`forall match`”, “`exp terminates`”, “`exp1 == exp2`”, 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 $\text{Dom } VE$ 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 $E + \text{Rec } VE$, where

$$\text{Rec} : \text{VarEnv} \rightarrow \text{VarEnv}$$

is defined as follows:

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

The effect is that, before application of $(match, E, VE)$ to v , the closure values in $\text{Ran } VE$ 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 \perp . 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 8.8 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 $AI : \text{Val} \rightarrow \text{Bit}$ 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).

$AI(v) = \top$ means that the value v neither directly nor indirectly depends on specification constructs, $AI(v) = \perp$ 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 $AI(E, exp) \wedge sp$ of $s \leq AI(v) \wedge sp$ 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 $\widehat{\text{Rec}} : \text{VarEnv} \rightarrow \text{VarEnv}$, defined as follows:

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

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

The operation $\widehat{\text{Rec}}$ 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 $\text{AI}(v) = \perp$ if v either contains **Incomplete** or is a closure that contains specification constructs or refers to incompletely defined identifiers:

$$\begin{aligned}
\text{AI} : \text{Val} &\rightarrow \text{Bit} \\
\text{AI}(\text{Incomplete}) &= \perp \\
\text{AI}(v) &= \top \quad \text{if } v \in \text{SVal} \uplus \text{BasVal} \uplus \text{Con} \uplus \text{ExName} \\
\text{AI}(\text{con}, v) &= \text{AI}(v) \\
\text{AI}(\text{en}, v) &= \text{AI}(v) \\
\text{AI}(r) &= \bigwedge_{\text{lab} \in \text{Dom } r} \text{AI}(r(\text{lab})) \\
\text{AI}(\text{match}, E, VE) &= \text{AI}(E + \widehat{\text{Rec}} VE, \text{match})
\end{aligned}$$

The function AI on the right-hand side of the last equation has the form $\text{AI} : \text{Env} \times \text{Match} \rightarrow \text{Bit}$. Similarly, we have a function $\text{AI} : \text{Env} \times \text{Exp} \rightarrow \text{Bit}$ 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.

$$\begin{aligned}
\text{AI} : \text{Env} \times \text{Match} &\rightarrow \text{Bit} \\
\text{AI}(E, \text{mrule} \langle \mid \text{match} \rangle) &= \text{AI}(E, \text{mrule}) \langle \wedge \text{AI}(E, \text{match}) \rangle \\
\text{AI} : \text{Env} \times \text{Mrule} &\rightarrow \text{Bit} \\
\text{AI}(E, \text{pat} \Rightarrow \text{exp}) &= \text{AI}(E + \text{AI}(\text{pat}), \text{exp})
\end{aligned}$$

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.

$$\begin{aligned}
\text{AI} : \text{Env} \times \text{Exp} &\rightarrow \text{Bit} \\
\text{AI}(E, \text{atexp}) &= \text{AI}(E, \text{atexp}) \\
\text{AI}(E, \text{exp atexp}) &= \text{AI}(E, \text{exp}) \wedge \text{AI}(E, \text{atexp}) \\
\text{AI}(E, \text{exp}_1 \bullet == \text{exp}_2 \bullet) &= \perp \\
\text{AI}(E, \text{exists } \text{match} \bullet) &= \perp \\
\text{AI}(E, \text{forall } \text{match} \bullet) &= \perp \\
\text{AI}(E, \text{exp} \bullet \text{terminates}) &= \perp \\
\text{AI}(E, \text{exp handle } \text{match}) &= \text{AI}(E, \text{exp}) \wedge \text{AI}(E, \text{match}) \\
\text{AI}(E, \text{raise } \text{exp}) &= \text{AI}(E, \text{exp}) \\
\text{AI}(E, \text{fn } \text{match}) &= \text{AI}(E, \text{match})
\end{aligned}$$

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.

$$\begin{aligned}
& \text{AI} : \text{Env} \times \text{AtExp} \rightarrow \text{Bit} \\
& \text{AI}(E, \text{scon}) &= \top \\
& \text{AI}(E, \text{longvar}) &= \text{AI}(E(\text{longvar})) \\
& \text{AI}(E, \text{longcon}) &= \text{AI}(E(\text{longcon})) \\
& \text{AI}(E, \text{longexcon}) &= \text{AI}(E(\text{longexcon})) \\
& \text{AI}(E, \{ \langle \text{exprow} \rangle \}) &= \top \langle \wedge \text{AI}(E, \text{exprow}) \rangle \\
& \text{AI}(E, \text{let } \text{dec} \text{ in } \text{exp} \text{ end}) &= \text{sp} \wedge \text{AI}(E + E', \text{exp}) \\
& &\text{where } (\text{sp}, E') = \text{AI}(E, \text{dec}) \\
& \text{AI}(E, (\text{exp})) &= \text{AI}(E, \text{exp}) \\
& \text{AI}(E, ?) &= \perp
\end{aligned}$$

The abstract interpretation of a `let`-expression `let dec in exp end` is \perp 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.

$$\begin{aligned}
& \text{AI} : \text{Env} \times \text{ExpRow} \rightarrow \text{Bit} \\
& \text{AI}(E, \text{lab} = \text{exp} \langle , \text{exprow} \rangle) = \text{AI}(E, \text{exp}) \langle \wedge \text{AI}(E, \text{exprow}) \rangle
\end{aligned}$$

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 $\text{AI}(v) = \top$ would do.

$$\begin{aligned}
& \text{AI} : \text{Pat} \rightarrow \text{VarEnv} \\
& \text{AI}(\text{atpat}) &= \text{AI}(\text{atpat}) \\
& \text{AI}(\text{longcon } \text{atpat}) &= \text{AI}(\text{atpat}) \\
& \text{AI}(\text{longexcon } \text{atpat}) &= \text{AI}(\text{atpat}) \\
& \text{AI}(\text{var as } \text{pat}) &= \{id \mapsto 1\} + \text{AI}(\text{pat}), \text{ where } \text{var} = id^{\mathbf{V}}
\end{aligned}$$

$$\begin{aligned}
& \text{AI} : \text{AtPat} \rightarrow \text{VarEnv} \\
& \text{AI}(-) &= \{\} \\
& \text{AI}(\text{scon}) &= \{\} \\
& \text{AI}(\text{var}) &= \{id \mapsto 1\}, \text{ where } \text{var} = id^{\mathbf{V}} \\
& \text{AI}(\text{longcon}) &= \{\} \\
& \text{AI}(\text{longexcon}) &= \{\} \\
& \text{AI}(\{ \langle \text{patrow} \rangle \}) &= \{\} \langle + \text{AI}(\text{patrow}) \rangle \\
& \text{AI}((\text{pat})) &= \text{AI}(\text{pat})
\end{aligned}$$

$$\begin{aligned}
& \text{AI} : \text{PatRow} \rightarrow \text{VarEnv} \\
& \text{AI}(\dots) &= \{\} \\
& \text{AI}(\text{lab} = \text{pat} \langle , \text{patrow} \rangle) &= \text{AI}(\text{pat}) \langle + \text{AI}(\text{patrow}) \rangle
\end{aligned}$$

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

$$\begin{aligned}
& \text{AI} : \text{Env} \times \text{Dec} \rightarrow \text{Bit} \times \text{Env} \\
& \text{AI}(E, \text{val } \text{valbind}) = (sp, VE \text{ in Env}) \\
& \quad \text{where } (sp, VE) = \text{AI}(E, \text{valbind}) \\
& \text{AI}(E, \text{datatype } \text{datbind}) = (\top, \text{AI}(\text{datbind}) \text{ in Env}) \\
& \text{AI}(E, \text{abstype } \text{datbind with } \text{dec end}) = (sp, (VE \text{ in Env}) + E') \\
& \quad \text{where } \begin{cases} VE & = \text{AI}(\text{datbind}) \\ (sp, E') & = \text{AI}(E + VE, \text{dec}) \end{cases} \\
& \text{AI}(E, \text{exception } \text{exbind}) = (\top, \text{AI}(\text{exbind}) \text{ in Env}) \\
& \text{AI}(E, \text{local } \text{dec}_1 \text{ in } \text{dec}_2 \text{ end}) = (sp_1 \wedge sp_2, E_2) \\
& \quad \text{where } \begin{cases} (sp_1, E_1) & = \text{AI}(E, \text{dec}_1) \\ (sp_2, E_2) & = \text{AI}(E + E_1, \text{dec}_2) \end{cases} \\
& \text{AI}(E, \text{open } \text{longstrid}_1 \cdots \text{longstrid}_n) = (\top, E_1 + \cdots + E_n) \\
& \quad \text{where } E_i = E(\text{longstrid}_i), 1 \leq i \leq n \\
& \text{AI}(E, \text{dec}_1 \langle ; \rangle \text{dec}_2) = (sp_1 \wedge sp_2, E_1 + E_2) \\
& \quad \text{where } \begin{cases} (sp_1, E_1) & = \text{AI}(E, \text{dec}_1) \\ (sp_2, E_2) & = \text{AI}(E + E_1, \text{dec}_2) \end{cases} \\
& \text{AI}(E, \quad) = (\top, \{\}) \\
& \text{AI} : E \times \text{ValBind} \rightarrow \text{Bit} \times \text{VarEnv} \\
& \text{AI}(E, \text{pat} = \text{exp} \langle \text{and } \text{valbind} \rangle) = (\text{AI}(E, \text{exp}) \langle \wedge sp \rangle, \text{AI}(\text{pat}) \langle + VE \rangle) \\
& \quad \text{where } (sp, VE) = \text{AI}(E, \text{valbind}) \\
& \text{AI}(E, \text{rec } \text{valbind}) = (sp, VE) \\
& \quad \text{where } (sp, VE) = \text{AI}(E + VE, \text{valbind})
\end{aligned}$$

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

$$\begin{aligned}
& \text{AI} : \text{DatBind} \rightarrow \text{VarEnv} \\
& \text{AI}(\text{tyvarseq } \text{tycon} = \text{conbind} \langle \text{and } \text{datbind} \rangle) = \text{AI}(\text{conbind}) \langle + \text{AI}(\text{datbind}) \rangle \\
& \text{AI} : \text{ConBind} \rightarrow \text{VarEnv} \\
& \text{AI}(\text{con} \langle | \text{conbind} \rangle) = \{id \mapsto 1\} \langle + \text{AI}(\text{conbind}) \rangle \\
& \quad \text{where } \text{con} = id^{\mathbf{V}} \\
& \text{AI} : \text{ExBind} \rightarrow \text{VarEnv} \\
& \text{AI}(\text{excon} \langle \text{and } \text{exbind} \rangle) = \{id \mapsto 1\} \langle + \text{AI}(\text{exbind}) \rangle \\
& \quad \text{where } \text{excon} = id^{\mathbf{e}} \\
& \text{AI}(\text{excon} = \text{longexcon} \langle \text{and } \text{exbind} \rangle) = \{id \mapsto 1\} \langle + \text{AI}(\text{exbind}) \rangle \\
& \quad \text{where } \text{excon} = id^{\mathbf{e}}
\end{aligned}$$

6.7 Inference Rules

The semantic rules allow sentences of the form

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

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

$$\frac{A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1 \quad A_2 \vdash \textit{phrase}_2 \Rightarrow A'_2 \quad \dots \quad \dots \quad A_n \vdash \textit{phrase}_n \Rightarrow A'_n}{A \vdash \textit{phrase} \Rightarrow A'}$$

then the full form is intended to be

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

(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

$$\frac{s_1, A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \dots \quad s_n, A_n \vdash \textit{phrase}_n \Rightarrow A'_n, s'_n}{s, A \vdash \textit{phrase} \Rightarrow A', s'}$$

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

$$\frac{s_1, A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \dots \quad s_k, A_k \vdash \textit{phrase}_k \Rightarrow p', s'}{s, A \vdash \textit{phrase} \Rightarrow p', s'}$$

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

⁶There 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 $x/\text{Incomplete}$); furthermore, we extend environment modification to allow for failure as follows:

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

Atomic Expressions

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

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

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

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

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

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

$$\frac{\langle E \vdash \text{exprow} \Rightarrow r \rangle}{E \vdash \{ \langle \text{exprow} \rangle \} \Rightarrow \{ \} \langle + r \rangle \text{ in Val}} \quad (107)$$

$$\frac{E \vdash \text{dec} \Rightarrow E' \quad E + E' \vdash \text{exp} \Rightarrow v}{E \vdash \text{let dec in exp end} \Rightarrow v} \quad (108)$$

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash (\text{exp}) \Rightarrow v} \quad (109)$$

$$\frac{}{s, E \vdash ? \Rightarrow [\text{NoCode}], s_{\perp}} \quad (109.1)$$

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.

Expression Rows

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

$$\frac{E \vdash \text{exp} \Rightarrow v \quad \langle E \vdash \text{exprow} \Rightarrow r \rangle}{E \vdash \text{lab} = \text{exp} \langle , \text{exprow} \rangle \Rightarrow \{\text{lab} \mapsto v\} \langle + r \rangle} \quad (110)$$

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

Expressions

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

$$\frac{E \vdash \text{atexp} \Rightarrow v}{E \vdash \text{atexp} \Rightarrow v} \quad (111)$$

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

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

$$\text{Deleted (ref application rule)} \quad (114)$$

$$\text{Deleted (:= application rule)} \quad (115)$$

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

$$\frac{E \vdash \text{exp} \Rightarrow (\text{match}, E', VE) \quad E \vdash \text{atexp} \Rightarrow v \quad E' + \text{Rec } VE, v \vdash \text{match} \Rightarrow v'}{E \vdash \text{exp atexp} \Rightarrow v'} \quad (117)$$

$$\frac{E \vdash \text{exp} \Rightarrow (\text{match}, E', VE) \quad E \vdash \text{atexp} \Rightarrow v \quad E' + \text{Rec } VE, v \vdash \text{match} \Rightarrow \text{FAIL}}{E \vdash \text{exp atexp} \Rightarrow [\text{Match}]} \quad (118)$$

$$\frac{}{s, E \vdash \text{exp}_1^\bullet == \text{exp}_2^\bullet \Rightarrow [\text{NoCode}], s_\perp} \quad (118.1)$$

$$\frac{}{s, E \vdash \text{exists match}^\bullet \Rightarrow [\text{NoCode}], s_\perp} \quad (118.2)$$

$$\frac{}{s, E \vdash \text{forall match}^\bullet \Rightarrow [\text{NoCode}], s_\perp} \quad (118.3)$$

$$\frac{}{s, E \vdash \mathit{exp}^\bullet \text{ terminates} \Rightarrow [\text{NoCode}], s_\perp} \quad (118.4)$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v}{E \vdash \mathit{exp} \text{ handle } \mathit{match} \Rightarrow v} \quad (119)$$

$$\frac{E \vdash \mathit{exp} \Rightarrow [e] \quad e \neq \text{NoCode} \quad E, e \vdash \mathit{match} \Rightarrow v}{E \vdash \mathit{exp} \text{ handle } \mathit{match} \Rightarrow v} \quad (120)$$

$$\frac{E \vdash \mathit{exp} \Rightarrow [e] \quad e \neq \text{NoCode} \quad E, e \vdash \mathit{match} \Rightarrow \text{FAIL}}{E \vdash \mathit{exp} \text{ handle } \mathit{match} \Rightarrow [e]} \quad (121)$$

$$\frac{E \vdash \mathit{exp} \Rightarrow [\text{NoCode}]}{E \vdash \mathit{exp} \text{ handle } \mathit{match} \Rightarrow [\text{NoCode}]} \quad (121.1)$$

$$\frac{E \vdash \mathit{exp} \Rightarrow e}{E \vdash \text{raise } \mathit{exp} \Rightarrow [e]} \quad (122)$$

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

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 $(\perp, \text{ens of } s)$.

(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

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

$$\frac{E, v \vdash \text{mrule} \Rightarrow v'}{E, v \vdash \text{mrule} \langle \mid \text{match} \rangle \Rightarrow v'} \quad (124)$$

$$\frac{E, v \vdash \text{mrule} \Rightarrow \text{FAIL}}{E, v \vdash \text{mrule} \Rightarrow \text{FAIL}} \quad (125)$$

$$\frac{E, v \vdash \text{mrule} \Rightarrow \text{FAIL} \quad E, v \vdash \text{match} \Rightarrow v'/\text{FAIL}}{E, v \vdash \text{mrule} \mid \text{match} \Rightarrow v'/\text{FAIL}} \quad (126)$$

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

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

$$\frac{E, v \vdash \text{pat} \Rightarrow VE \quad E + VE \vdash \text{exp} \Rightarrow v'}{E, v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow v'} \quad (127)$$

$$\frac{E, v \vdash \text{pat} \Rightarrow \text{FAIL}}{E, v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \text{FAIL}} \quad (128)$$

Declarations

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

$$\frac{E \vdash \text{valbind} \Rightarrow VE}{E \vdash \text{val } \text{valbind} \Rightarrow VE \text{ in Env}} \quad (129)$$

$$\frac{\vdash \text{datbind} \Rightarrow VE}{E \vdash \text{datatype } \text{datbind} \Rightarrow VE \text{ in Env}} \quad (129.5)$$

$$\frac{\vdash \text{datbind} \Rightarrow VE \quad E + VE \vdash \text{dec} \Rightarrow E'}{E \vdash \text{abstype } \text{datbind} \text{ with } \text{dec} \text{ end} \Rightarrow E'} \quad (129.6)$$

$$\frac{E \vdash \text{exbind} \Rightarrow VE}{E \vdash \text{exception } \text{exbind} \Rightarrow VE \text{ in Env}} \quad (130)$$

$$\frac{E \vdash \text{dec}_1 \Rightarrow E_1 \quad E + E_1 \vdash \text{dec}_2 \Rightarrow E_2}{E \vdash \text{local } \text{dec}_1 \text{ in } \text{dec}_2 \text{ end} \Rightarrow E_2} \quad (131)$$

$$\frac{E(\text{longstrid}_1) = E_1 \quad \cdots \quad E(\text{longstrid}_n) = E_n}{E \vdash \text{open } \text{longstrid}_1 \cdots \text{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \quad (132)$$

$$\overline{E \vdash} \Rightarrow \{\} \text{ in Env} \quad (133)$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \quad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash dec_1 \langle ; \rangle dec_2 \Rightarrow E_1 + E_2} \quad (134)$$

Value Bindings

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

$$\frac{E \vdash exp \Rightarrow v \quad E, v \vdash pat \Rightarrow VE \quad \langle E \vdash valbind \Rightarrow VE' \rangle}{E \vdash pat = exp \langle \text{and valbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (135)$$

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

$$\frac{E \vdash exp \Rightarrow [e], e \neq \text{NoCode}}{E \vdash pat = exp \langle \text{and valbind} \rangle \Rightarrow [e]} \quad (135.2)$$

$$\frac{E \vdash exp \Rightarrow v \quad E, v \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat = exp \langle \text{and valbind} \rangle \Rightarrow [\text{Bind}]} \quad (136)$$

$$\frac{E \vdash exp \Rightarrow [\text{NoCode}] \quad E, \text{Incomplete} \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat = exp \langle \text{and valbind} \rangle \Rightarrow [\text{NoCode}]} \quad (136.1)$$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \text{rec valbind} \Rightarrow \text{Rec } VE} \quad (137)$$

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

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

$$\frac{\vdash conbind \Rightarrow VE \quad \langle \vdash datbind \Rightarrow VE' \rangle}{\vdash tyvarseq tycon = conbind \langle \text{and datbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (137.1)$$

Constructor Bindings

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

$$\frac{\text{con} = id^c \quad \langle \vdash \text{conbind} \Rightarrow VE \rangle}{\vdash \text{con} \langle | \text{conbind} \rangle \Rightarrow \{id \mapsto \text{con}\} \langle + VE \rangle} \quad (137.2)$$

Exception Bindings

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

In the SML definition, sentences for exception bindings have the more general form $E \vdash \text{exbind} \Rightarrow EE/p$. 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.

$$\frac{\begin{array}{l} en = \min(\text{ExName} \setminus (\text{ens of } s)) \quad s' = s + \{en\} \\ \text{excon} = id^e \quad \langle s', E \vdash \text{exbind} \Rightarrow VE, s'' \rangle \end{array}}{s, E \vdash \text{excon} \langle \text{and exbind} \rangle \Rightarrow \{id \mapsto en\} \langle + VE \rangle, s' \langle ' \rangle} \quad (138)$$

$$\frac{E(\text{longexcon}) = en \quad \text{excon} = id^e \quad \langle E \vdash \text{exbind} \Rightarrow VE \rangle}{E \vdash \text{excon} = \text{longexcon} \langle \text{and exbind} \rangle \Rightarrow \{id \mapsto en\} \langle + VE \rangle} \quad (139)$$

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 259 and 260.

Atomic Patterns

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

$$\overline{E, v \vdash _ \Rightarrow \{ \}} \quad (140)$$

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

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

$$\frac{\text{var} = id^v}{E, v \vdash \text{var} \Rightarrow \{id \mapsto v\}} \quad (143)$$

$$\frac{E(\text{longcon}) = con \quad con = v}{E, v \vdash \text{longcon} \Rightarrow \{ \}} \quad (144)$$

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

$$\frac{E(\text{longexcon}) = en \quad en = v}{E, v \vdash \text{longexcon} \Rightarrow \{\}} \quad (146)$$

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

$$\frac{v = \{\}\langle+r\rangle \text{ in Val} \quad \langle E, r \vdash \text{patrow} \Rightarrow VE/\text{FAIL} \rangle}{E, v \vdash \{ \langle \text{patrow} \rangle \} \Rightarrow \{\}\langle+VE/\text{FAIL} \rangle} \quad (148)$$

$$\frac{v = \text{Incomplete} \quad \langle E, \text{Incomplete} \vdash \text{patrow} \Rightarrow VE/\text{FAIL} \rangle}{E, v \vdash \{ \langle \text{patrow} \rangle \} \Rightarrow \{\}\langle+VE/\text{FAIL} \rangle} \quad (148.1)$$

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

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 $\text{pat} = ? \langle \text{and valbind} \rangle$ (or $\text{pat} = \text{exp} \langle \text{and valbind} \rangle$ 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

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

$$\overline{E, r \vdash \dots \Rightarrow \{\}}$$
 (150)

$$\overline{E, \text{Incomplete} \vdash \dots \Rightarrow \{\}}$$
 (150.1)

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

$$\frac{E, \text{Incomplete} \vdash \text{pat} \Rightarrow \text{FAIL}}{E, \text{Incomplete} \vdash \text{lab} = \text{pat} \langle \ , \text{patrow} \rangle \Rightarrow \text{FAIL}}$$
 (151.1)

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

$$\frac{E, \text{Incomplete} \vdash \text{pat} \Rightarrow VE \quad \langle E, \text{Incomplete} \vdash \text{patrow} \Rightarrow VE' / \text{FAIL} \rangle}{E, \text{Incomplete} \vdash \text{lab} = \text{pat} \langle \ , \text{patrow} \rangle \Rightarrow VE \langle + VE' / \text{FAIL} \rangle}$$
 (152.1)

Comments:

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

Patterns

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

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

$$\frac{E(\text{longcon}) = \text{con} \quad v = (\text{con}, v') \quad E, v' \vdash \text{atpat} \Rightarrow VE / \text{FAIL}}{E, v \vdash \text{longcon atpat} \Rightarrow VE / \text{FAIL}}$$
 (154)

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

$$\frac{E(\text{longexcon}) = \text{en} \quad v = (\text{en}, v') \quad E, v' \vdash \text{atpat} \Rightarrow VE / \text{FAIL}}{E, v \vdash \text{longexcon atpat} \Rightarrow VE / \text{FAIL}}$$
 (156)

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

Deleted (**ref** application rule) (158)

$$\frac{var = id^{\mathbf{v}} \quad E, v \vdash pat \Rightarrow VE/FAIL}{E, v \vdash var \text{ as } pat \Rightarrow \{id \mapsto v\} + VE/FAIL} \quad (159)$$

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* $I \in \text{Int}$ 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

$$\text{Inter} : \begin{cases} \text{Env} \rightarrow \text{Int} \\ \text{VarEnv} \rightarrow \text{StatEnv} \end{cases}$$

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

Figure 16: Further Semantic Objects

as follows:

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

where

$$\begin{aligned}
IE &= \{strid \mapsto \text{Inter } E ; SE(strid) = E\} \\
\text{Inter } VE &= \{id \mapsto \mathbf{v} ; id \in \text{Dom } VE\}
\end{aligned}$$

Notice that *all* identifiers are assigned status \mathbf{v} in $\text{Inter } VE$. 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

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

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

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

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

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

as follows:

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

where

$$\begin{aligned}
SE &= \{strid_1 \mapsto E_1, \dots, strid_n \mapsto E_n\} \\
VE &= \{id \mapsto \text{Incomplete} ; StE(id) = \mathbf{v}\} + \\
&\quad \{id \mapsto id^c ; StE(id) = \mathbf{c}\} + \\
&\quad \{id_1 \mapsto en_1, \dots, id_k \mapsto en_k ; StE(id_j) = \mathbf{e}, 1 \leq j \leq k\} \\
ens' &= ens_n \cup \{en_1, \dots, en_k\}
\end{aligned}$$

where

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

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

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

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

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

as follows:

```

TrivStrExp(IE, StE) = struct
    structure strid1 = TrivStrExp(IE(strid1))
        ...
        and stridn = TrivStrExp(IE(stridn))
    val var1 = ? and ... and varp = ?
    exception excon1 and ... and exconm
    datatype dummy = con1 | ... | conr
end

```

where:

$$\begin{aligned}
 \{strid_1, \dots, strid_n\} &= \text{Dom } IE \\
 \{var_1, \dots, var_p\} &= \{id^V \mid id \in \text{Dom } StE ; StE(id) = v\} \\
 \{excon_1, \dots, excon_m\} &= \{id^e \mid id \in \text{Dom } StE ; StE(id) = e\} \\
 \{con_1, \dots, con_r\} &= \{id^C \mid id \in \text{Dom } StE ; StE(id) = c\}
 \end{aligned}$$

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

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

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

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

$$\frac{B \vdash \textit{strdec} \Rightarrow E}{B \vdash \textbf{struct } \textit{strdec} \textbf{ end} \Rightarrow E} \quad (160)$$

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

$$\frac{B(\textit{funid}) = (\textit{strid} : I, \textit{strexp}' : I', B') \quad B \vdash \textit{strexp} \Rightarrow E \quad B' + \{\textit{strid} \mapsto E \downarrow I\} \vdash \textit{strexp}' \Rightarrow E'}{B \vdash \textit{funid} (\textit{strexp}) \Rightarrow E' \downarrow I'} \quad (162)$$

$$\frac{B \vdash \text{strdec} \Rightarrow E \quad B + E \vdash \text{strex} \Rightarrow E'}{B \vdash \text{let strdec in strex end} \Rightarrow E'} \quad (163)$$

Comments:

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

Structure-level Declarations

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

$$\frac{E \text{ of } B \vdash \text{dec} \Rightarrow E'}{B \vdash \text{dec} \Rightarrow E'} \quad (164)$$

$$\frac{}{s, B \vdash \text{axiom } ax \Rightarrow \{\} \text{ in Env, } s} \quad (164.1)$$

$$\frac{B \vdash \text{strbind} \Rightarrow SE}{B \vdash \text{structure strbind} \Rightarrow SE \text{ in Env}} \quad (165)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{local strdec}_1 \text{ in strdec}_2 \text{ end} \Rightarrow E_2} \quad (166)$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env}} \quad (167)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{strdec}_1 \langle ; \rangle \text{ strdec}_2 \Rightarrow E_1 + E_2} \quad (168)$$

Structure Bindings

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

$$\frac{B \vdash \text{sglstrbind} \Rightarrow SE \quad \langle B \vdash \text{strbind} \Rightarrow SE' \rangle}{B \vdash \text{sglstrbind} \langle \text{and strbind} \rangle \Rightarrow SE \langle +SE' \rangle} \quad (169)$$

Single Structure Bindings

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

$$\frac{B \vdash \text{strex} \Rightarrow E \quad \text{Inter } B \vdash \text{psigexp} \Rightarrow I}{B \vdash \text{strid} : \text{psigexp} = \text{strex} \Rightarrow \{\text{strid} \mapsto E \downarrow I\}} \quad (169.1)$$

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

$$\frac{B \vdash strexp \Rightarrow E}{B \vdash strid = strexp \Rightarrow \{strid \mapsto E\}} \quad (169.3)$$

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

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

$$\frac{IB \vdash spec \Rightarrow I}{IB \vdash \mathbf{sig\ spec\ end} \Rightarrow I} \quad (170)$$

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

Principal Signatures

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

$$\frac{IB \vdash sigexp \Rightarrow I}{IB \vdash sigexp \Rightarrow I} \quad (171.1)$$

Signature Declarations

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

$$\frac{IB \vdash sigbind \Rightarrow G}{IB \vdash \mathbf{signature\ sigbind} \Rightarrow G} \quad (172)$$

$$\overline{IB \vdash \quad \Rightarrow \{\}} \quad (173)$$

$$\frac{IB \vdash sigdec_1 \Rightarrow G_1 \quad IB + G_1 \vdash sigdec_2 \Rightarrow G_2}{IB \vdash sigdec_1 \langle ; \rangle sigdec_2 \Rightarrow G_1 + G_2} \quad (174)$$

Signature Bindings

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

$$\frac{IB \vdash sigexp \Rightarrow I \quad \langle IB \vdash sigbind \Rightarrow G \rangle}{IB \vdash sigid = psigexp \langle \mathbf{and\ sigbind} \rangle \Rightarrow \{sigid \mapsto I\} \langle + G \rangle} \quad (175)$$

Specifications

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

$$\frac{\vdash valdesc \Rightarrow StE}{IB \vdash \mathbf{val} \, valdesc \Rightarrow StE \text{ in Int}} \quad (176)$$

$$\frac{\vdash datdesc \Rightarrow StE}{IB \vdash \mathbf{datatype} \, datdesc \Rightarrow StE \text{ in Int}} \quad (176.1)$$

$$\frac{\vdash exdesc \Rightarrow StE}{IB \vdash \mathbf{exception} \, exdesc \Rightarrow StE \text{ in Int}} \quad (177)$$

$$\frac{IB \vdash strdesc \Rightarrow IE}{IB \vdash \mathbf{structure} \, strdesc \Rightarrow IE \text{ in Int}} \quad (178)$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \quad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash \mathbf{local} \, spec_1 \text{ in } spec_2 \, \mathbf{end} \Rightarrow I_2} \quad (179)$$

$$\frac{IB(longstrid_1) = I_1 \quad \dots \quad IB(longstrid_n) = I_n}{IB \vdash \mathbf{open} \, longstrid_1 \, \dots \, longstrid_n \Rightarrow I_1 + \dots + I_n} \quad (180)$$

$$\frac{IB(sigid_1) = I_1 \quad \dots \quad IB(sigid_n) = I_n}{IB \vdash \mathbf{include} \, sigid_1 \, \dots \, sigid_n \Rightarrow I_1 + \dots + I_n} \quad (181)$$

$$\frac{}{IB \vdash \quad \Rightarrow \{ \} \text{ in Int}} \quad (182)$$

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

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

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

$$\frac{var = id^V \quad \langle \vdash valdesc \Rightarrow StE \rangle}{\vdash var \langle \mathbf{and} \, valdesc \rangle \Rightarrow \{ id \mapsto v \} \langle +StE \rangle} \quad (184)$$

Datatype Descriptions

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

$$\frac{\vdash condesc \Rightarrow StE \quad \langle \vdash datdesc \Rightarrow StE' \rangle}{\vdash tyvarseq \, tycon = condesc \langle \mathbf{and} \, datdesc \rangle \Rightarrow StE \langle +StE' \rangle} \quad (184.1)$$

Constructor Descriptions

$$\boxed{\vdash \text{condesc} \Rightarrow \text{StE}}$$

$$\frac{\text{con} = \text{id}^c \quad \langle \vdash \text{condesc} \Rightarrow \text{StE} \rangle}{\vdash \text{con} \langle | \text{condesc} \rangle \Rightarrow \{ \text{id} \mapsto \mathbf{c} \} \langle + \text{StE} \rangle} \quad (184.2)$$

Exception Descriptions

$$\boxed{\vdash \text{exdesc} \Rightarrow \text{StE}}$$

$$\frac{\text{excon} = \text{id}^e \quad \langle \vdash \text{exdesc} \Rightarrow \text{StE} \rangle}{\vdash \text{excon} \langle \text{and exdesc} \rangle \Rightarrow \{ \text{id} \mapsto \mathbf{e} \} \langle + \text{StE} \rangle} \quad (185)$$

Structure Descriptions

$$\boxed{IB \vdash \text{strdesc} \Rightarrow IE}$$

$$\frac{IB \vdash \text{sigexp} \Rightarrow I \quad \langle IB \vdash \text{strdesc} \Rightarrow IE \rangle}{IB \vdash \text{strid} : \text{sigexp} \langle \text{and strdesc} \rangle \Rightarrow \{ \text{strid} \mapsto I \} \langle + IE \rangle} \quad (186)$$

Functor Bindings

$$\boxed{B \vdash \text{funbind} \Rightarrow F}$$

$$\frac{\text{Inter } B \vdash \text{psigexp} \Rightarrow I \quad \text{Inter } B + \{ \text{strid} \mapsto I \} \vdash \text{psigexp}' \Rightarrow I' \quad \langle B \vdash \text{funbind} \Rightarrow F \rangle}{B \vdash \text{funid} (\text{strid} : \text{psigexp}) : \text{psigexp}' = \text{strex} \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (\text{strid} : I, \text{strex} : I', B) \} \langle + F \rangle} \quad (187)$$

$$\frac{\text{Inter } B \vdash \text{psigexp} \Rightarrow I \quad \text{Inter } B + \{ \text{strid} \mapsto I \} \vdash \text{psigexp}' \Rightarrow I' \quad \langle B \vdash \text{funbind} \Rightarrow F \rangle}{B \vdash \text{funid} (\text{strid} : \text{psigexp}) : \text{psigexp}' = ? \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (\text{strid} : I, \text{TrivStrExp } I' : I', B) \} \langle + F \rangle} \quad (187.1)$$

Functor Declarations

$$\boxed{B \vdash \text{fundec} \Rightarrow F}$$

$$\frac{B \vdash \text{funbind} \Rightarrow F}{B \vdash \text{functor funbind} \Rightarrow F} \quad (188)$$

$$\overline{B \vdash \quad \Rightarrow \{ \}} \quad (189)$$

$$\frac{B \vdash \text{fundec}_1 \Rightarrow F_1 \quad B + F_1 \vdash \text{fundec}_2 \Rightarrow F_2}{B \vdash \text{fundec}_1 \langle ; \rangle \text{fundec}_2 \Rightarrow F_1 + F_2} \quad (190)$$

Top-level Declarations

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

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

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

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

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 $\text{Dom } \vartheta = \alpha^{(k)}$ 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.) $\text{Clos}_C(v, \tau) = \forall \alpha^{(k)}.(v, \tau)$, 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. $\text{Clos}_{FE}VE$ 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

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

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 six 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; a natural number n for counting occurrences of $?$; and a flag l which will be used later in the module semantics. We write $s \oplus n$ for the state in which n of s is *replaced* by n . Similarly to the dynamic semantics, we write s_{\perp} and s_{\top} for the state s modified by setting l to \perp or \top , respectively.

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 $\text{Replace} : (\text{Phrase} \times \text{QInt} \times \mathcal{N}) \rightarrow (\text{Phrase} \times \mathcal{N})$, 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:

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

On other syntactical phrases, Replace is defined completely analogously, following the schema of let -expressions for all composed phrases. $\text{Replace}(phrase, QI, n)$ 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 $\text{Comp} : \text{Env} \times \text{ValTemp} \rightarrow \text{VarEnv}$ for completing environments as follows: $\text{Comp}(E, VT) = VE$ 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 $VE(id') = \forall \alpha^{(k)}. (id^{\mathcal{C}}, \tau)$.
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 $\text{Comp}(E \text{ of } FE, VT \text{ of } s)$ as $\text{Comp}(FE, s)$.

The intention is to enable each value “belonging to a type” to be obtained by verifying some expression in $\text{Comp}(FE, s)$.

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 interpretation 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:

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

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 a type interpretation TI and a type realisation 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 $(\varphi_{\text{Ty}} \text{ of } s)^{\#} \circ s^{\#}$ 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 Fresh Type Names

The static semantics ensures that the type names chosen for two different data-type definitions are different whenever these types “exist” at the same point in a program. For the purposes of verification this insurance is not secure enough, because the earlier described Comp functions tears down all scope barriers. To deal with that properly we rename type names in a certain way. For this purpose we employ a function $\text{Fresh} : \text{TyNameSet} \times \text{State} \rightarrow \text{TyRea}$ which is specified as follows: let $\text{Fresh}(T, s) = \varphi_{\text{Ty}}$ then the following properties hold:

- φ_{Ty} is injective and $\varphi_{\text{Ty}}^{-1} \in \text{TyRea}$
- $\text{Supp}(\varphi_{\text{Ty}}) = T$
- $\text{Yield}(\varphi_{\text{Ty}}) \cap \text{tynames}(s) = \emptyset$.

8.7 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:

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

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:

$$\begin{aligned} \text{Stat}(SE, TE, VE) &= (\text{Stat } SE, TE, \text{Stat } VE) \\ \text{Stat } VE &= \text{Stat} \circ VE \\ \text{Stat}(\forall\alpha^{(k)}.(v, \tau)) &= \forall\alpha^{(k)}. \tau \\ \text{Stat } SE &= \{ \text{strid} \mapsto (m, \text{Stat } E) ; SE(\text{strid}) = (E, m) \} \end{aligned}$$

8.8 Relationship to Dynamic Semantics

If any expression exp^\bullet evaluates in the dynamic semantics to a value or packet, i.e. $s_{\text{DYN}}, E_{\text{DYN}} \vdash_{\text{DYN}} exp^\bullet \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, s'_{\text{DYN}}$ without changing the flag, i.e. sp of $s_{\text{DYN}} = sp$ of $s'_{\text{DYN}} = \top$, then for any s such that $s_{\text{DYN}} = \text{Dyn } s$ and for any M such that $E_{\text{DYN}} = \text{Dyn } M$ we also have $s, M \vdash exp^\bullet \Rightarrow v/p, s'$ such that $\text{Dyn}(v/p) = v_{\text{DYN}}/p_{\text{DYN}}$ and $\text{Dyn } s' = s'_{\text{DYN}}$, 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 verifying 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 .

8.9 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_{STAT} or \vdash_{DYN} , respectively.

8.10 Inference Rules

The semantic rules allow sentences of the form $s, A \vdash phrase \Rightarrow A', s'$ 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 $s, FE, \gamma \vdash exp_1^\bullet \equiv exp_2^\bullet \Rightarrow v$, comparison sentences $C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow v$ and approximation sentences $C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow v$. 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 8.9. 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 224–231 (equality and comparison rules).

Atomic Expressions

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

$$\frac{}{M \vdash scon \Rightarrow \text{val}(scon)} \quad (194)$$

$$\frac{FE(\text{longvar}) \succ (v, s^\#(\tau)) \quad \frac{FE(\text{longvar}) \succ (v', s^\#(\tau))}{\exists \vartheta. \vartheta(v) = v' \wedge \text{Dom } \vartheta \cap \text{tyvars } FE = \emptyset}}{s, (FE, \tau) \vdash \text{longvar} \Rightarrow v, s} \quad (195)$$

$$\frac{M(\text{longcon}) \succ (\text{con}, \tau)}{M \vdash \text{longcon} \Rightarrow \text{con}} \quad (196)$$

$$\frac{M(\text{longexcon}) = (\text{en}, \tau)}{M \vdash \text{longexcon} \Rightarrow \text{en}} \quad (197)$$

$$\frac{\langle (FE, \gamma) \vdash \text{exprow} \Rightarrow r \rangle}{(FE, \epsilon \langle \cdot \gamma \rangle) \vdash \{ \langle \text{exprow} \rangle \} \Rightarrow \{ \} \langle + r \rangle \text{ in Val}} \quad (198)$$

$$\frac{(FE, \gamma) \vdash \text{dec} \Rightarrow E' \quad (FE + E', \gamma') \vdash \text{exp} \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash \text{let dec in exp end} \Rightarrow v} \quad (199)$$

$$\frac{M \vdash \text{exp} \Rightarrow v}{M \vdash (\text{exp}) \Rightarrow v} \quad (200)$$

$$\frac{\begin{array}{l} n = n \text{ of } s \quad (QIE \text{ of } FE)(n) = \text{exp}^\bullet \\ s^\#(C) \vdash_{\text{STAT}} \text{exp}^\bullet \Rightarrow \tau', \emptyset, \gamma \quad s^\#(\tau) = \tau' \\ s \oplus (n + 1), (FE, \gamma) \vdash \text{exp}^\bullet \Rightarrow v, s' \end{array}}{s, (FE, (C, \tau)) \vdash ? \Rightarrow v, s'} \quad (201)$$

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 $?(x_1, \dots, x_n)$ 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

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

$$\frac{(FE, \gamma) \vdash exp \Rightarrow v \quad \langle (FE, \gamma') \vdash exprow \Rightarrow r \rangle}{(FE, \gamma \langle \cdot \gamma' \rangle) \vdash lab = exp \langle \cdot, exprow \rangle \Rightarrow \{lab \mapsto v\} \langle + r \rangle} \quad (202)$$

Expressions

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

$$\frac{M \vdash atexp \Rightarrow v}{M \vdash atexp \Rightarrow v} \quad (203)$$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow con \quad (FE, \gamma') \vdash atexp \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash exp atexp \Rightarrow (con, v)} \quad (204)$$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow en \quad (FE, \gamma') \vdash atexp \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash exp atexp \Rightarrow (en, v)} \quad (205)$$

$$\frac{(FE, \gamma) \vdash exp \Rightarrow b \quad (FE, \gamma') \vdash atexp \Rightarrow v \quad \text{APPLY}(b, \text{Dyn } v) = v'/p}{(FE, \gamma \cdot \gamma') \vdash exp atexp \Rightarrow v'/p} \quad (206)$$

$$\frac{s_1, (FE, \gamma) \vdash exp \Rightarrow (match, M, VE), s_2 \quad s_2, (FE, \gamma') \vdash atexp \Rightarrow v, s_3 \quad s_3 \oplus 0, M + \text{Rec } VE, v \vdash match \Rightarrow v', s_4}{s_1, (FE, \gamma \cdot \gamma') \vdash exp atexp \Rightarrow v', s_4 \oplus (n \text{ of } s_3)} \quad (207)$$

$$\frac{s_1, (FE, \gamma) \vdash exp \Rightarrow (match, M, VE), s_2 \quad s_2, (FE, \gamma') \vdash atexp \Rightarrow v, s_3 \quad s_3 \oplus 0, M + \text{Rec } VE, v \vdash match \Rightarrow \text{FAIL}, s_4}{s_1, (FE, \gamma \cdot \gamma') \vdash exp atexp \Rightarrow [\text{Match}], s_4 \oplus (n \text{ of } s_3)} \quad (208)$$

$$\frac{M \vdash exp \Rightarrow v}{M \vdash exp : ty \Rightarrow v} \quad (209)$$

$$\frac{\frac{\gamma \succ \gamma' = (C, \tau) \cdot \gamma''}{s, FE, \gamma' \vdash exp_1^\bullet \equiv exp_2^\bullet \Rightarrow v}}{s, (FE, \gamma) \vdash exp_1^\bullet == exp_2^\bullet \Rightarrow v, s} \quad (210)$$

Comments:

- (206) The reason for the application of “Dyn” is that the function APPLY operates on dynamic values and not verification values. This is largely irrelevant, because most basic values only operate on numbers and strings which are in both value classes. One exception though is the polymorphic equality =.
- (207),(208) To verificate *match* we have to reset the *nofs* 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 347.
- (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.

$$\frac{\left(\frac{\text{Comp}(FE, s) = VE \quad \gamma \succ \gamma_1 = (C, \tau) \cdot \gamma_2 \quad s^{\#\#}(\tau) = \tau'}{s^{\#\#}(C) + \text{Stat } VE \vdash_{\text{STAT}} \text{atexp}^\bullet \Rightarrow \tau', \emptyset, \gamma_3 \wedge} \quad \text{Dyn}(s, FE + VE) \vdash_{\text{DYN}} \text{atexp}^\bullet \Rightarrow v_{\text{DYN}}, (\top, \text{ens}) \wedge}{s, (FE + VE, \gamma_1 \cdot \gamma_3) \vdash (\text{fn } \text{match}^\bullet) \text{ atexp}^\bullet \Rightarrow \text{true}, s'} \right)}{s, (FE, \gamma) \vdash \text{exists } \text{match}^\bullet \Rightarrow \text{true}, s} \quad (211)$$

$$\frac{\text{Comp}(FE, s) = VE \quad \left(\frac{\gamma \succ \gamma_1 = (C, \tau) \cdot \gamma_2 \quad s^{\#\#}(C) + \text{Stat } VE \vdash_{\text{STAT}} \text{atexp}^\bullet \Rightarrow \tau', \emptyset, \gamma_3}{s^{\#\#}(\tau) = \tau' \quad \text{Dyn}(s, FE + VE) \vdash_{\text{DYN}} \text{atexp}^\bullet \Rightarrow v_{\text{DYN}}, (\top, \text{ens})} \right)}{\exists s'. s, (FE + VE, \gamma_1 \cdot \gamma_3) \vdash (\text{fn } \text{match}^\bullet) \text{ atexp}^\bullet \Rightarrow \text{false}, s'}}{s, (FE, \gamma) \vdash \text{exists } \text{match}^\bullet \Rightarrow \text{false}, s} \quad (212)$$

$$\frac{\text{Comp}(FE, s) = VE \quad \left(\frac{\gamma \succ \gamma_1 = (C, \tau) \cdot \gamma_2 \quad s^{\#\#}(C) + \text{Stat } VE \vdash_{\text{STAT}} \text{atexp}^\bullet \Rightarrow \tau', \emptyset, \gamma_3}{s^{\#\#}(\tau) = \tau' \quad \text{Dyn}(s, FE + VE) \vdash_{\text{DYN}} \text{atexp}^\bullet \Rightarrow v_{\text{DYN}}, (\top, \text{ens})} \right)}{\exists s'. s, (FE + VE, \gamma_1 \cdot \gamma_3) \vdash (\text{fn } \text{match}^\bullet) \text{ atexp}^\bullet \Rightarrow \text{true}, s'}}{s, (FE, \gamma) \vdash \text{forall } \text{match}^\bullet \Rightarrow \text{true}, s} \quad (213)$$

$$\frac{\left(\frac{\text{Comp}(FE, s) = VE \quad \gamma \succ \gamma_1 = (C, \tau) \cdot \gamma_2 \quad s^{\#\#}(\tau) = \tau'}{s^{\#\#}(C) + \text{Stat } VE \vdash_{\text{STAT}} \text{atexp}^\bullet \Rightarrow \tau', \emptyset, \gamma_3 \wedge} \quad \text{Dyn}(s, FE + VE) \vdash_{\text{DYN}} \text{atexp}^\bullet \Rightarrow v_{\text{DYN}}, (\top, \text{ens}) \wedge}{s, (FE + VE, \gamma_1 \cdot \gamma_3) \vdash (\text{fn } \text{match}^\bullet) \text{atexp}^\bullet \Rightarrow \text{false}, s'} \right)}{s, (FE, \gamma) \vdash \text{forall } \text{match}^\bullet \Rightarrow \text{false}, s} \quad (214)$$

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 $\text{Comp}(FE, s)$ 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.

The premises of these rules contain a rule themselves, i.e. these are higher-order rules.

(211), (214) The premise requires a witness *for each* type instance. This is not the same as requiring a witness for the most general type, because the body of the quantifier may be type-dependent; typically this situation occurs when quantifiers are nested.

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.

$$\frac{\text{Dyn}(s, M) \vdash_{\text{DYN}} \text{exp}^\bullet \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, (\top, \text{ens})}{s, M \vdash \text{exp}^\bullet \text{terminates} \Rightarrow \text{true}, s} \quad (215)$$

$$\frac{\text{Dyn}(s, M) \vdash_{\text{DYN}} \text{exp}^\bullet \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, (\perp, \text{ens})}{s, M \vdash \text{exp}^\bullet \text{terminates} \Rightarrow [\text{Abuse}], s} \quad (216)$$

$$\frac{s_{\text{DYN}}, E_{\text{DYN}} = \text{Dyn}(s, M) \quad \neg \exists v_{\text{DYN}}/p_{\text{DYN}}, s'_{\text{DYN}} \cdot s_{\text{DYN}}, E_{\text{DYN}} \vdash_{\text{DYN}} \text{exp}^\bullet \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, s'_{\text{DYN}}}{s, M \vdash \text{exp}^\bullet \text{terminates} \Rightarrow \text{false}, s} \quad (217)$$

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 \perp , 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 \perp , 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 verified using the dynamic semantics. This is necessary to avoid paradoxes caused by the non-existence premise of this rule.

$$\frac{(FE, \gamma) \vdash \text{exp} \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash \text{exp handle match} \Rightarrow v} \quad (218)$$

Comments:

(210)–(218) These rules are exempted from the exception convention.

$$\frac{(FE, \gamma) \vdash \text{exp} \Rightarrow [e] \quad e \neq \text{Abuse} \quad (FE, \gamma'), e \vdash \text{match} \Rightarrow v}{(FE, \gamma \cdot \gamma') \vdash \text{exp handle match} \Rightarrow v} \quad (219)$$

$$\frac{(FE, \gamma) \vdash \text{exp} \Rightarrow [e] \quad e \neq \text{Abuse} \quad (FE, \gamma'), e \vdash \text{match} \Rightarrow \text{FAIL}}{(FE, \gamma \cdot \gamma') \vdash \text{exp handle match} \Rightarrow [e]} \quad (220)$$

$$\frac{(FE, \gamma) \vdash \text{exp} \Rightarrow [\text{Abuse}]}{(FE, \gamma \cdot \gamma') \vdash \text{exp handle match} \Rightarrow [\text{Abuse}]} \quad (221)$$

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.

$$\frac{(FE, \gamma) \vdash \text{exp} \Rightarrow e}{(FE, \tau \cdot \gamma) \vdash \text{raise exp} \Rightarrow [e]} \quad (222)$$

$$\frac{\begin{array}{l} n = n \text{ of } s \quad M = ((E, (QIE, QIT)), (C, \tau) \cdot \gamma) \\ (match', n') = \text{Replace}(match, (QIE, QIT), n) \\ s^\#(C) \vdash_{\text{STAT}} match' \Rightarrow \tau', U, \gamma' \quad s^\#(\tau) = \tau' \\ f = \lambda k.k - n \quad QIE' = QIE \circ f \quad QIT' = QIT \circ f \\ M' = ((E, (QIE', QIT')), \gamma) \end{array}}{s, M \vdash \text{fn match} \Rightarrow (match, M', \{\}), s \oplus n'} \quad (223)$$

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 verifying 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

$$\boxed{s, FE, \gamma \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow v}$$

$$\frac{\begin{array}{l} \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_1 \Rightarrow v_{\text{DYN}}^1, s_{\text{DYN}}^1 \quad sp \text{ of } s_{\text{DYN}}^1 = \top \\ \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_2 \Rightarrow v_{\text{DYN}}^2, s_{\text{DYN}}^2 \quad sp \text{ of } s_{\text{DYN}}^2 = \top \\ VE = \text{Comp}(FE, s) \quad C' = C + \text{Stat } VE \quad E_{\text{DYN}} = \text{Dyn}(FE + VE) \\ C', s^{\#\#}(\tau), E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow v \end{array}}{s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow v} \quad (224)$$

$$\frac{\begin{array}{l} \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_1 \Rightarrow [e_{\text{DYN}}^1], s_{\text{DYN}}^1 \quad sp \text{ of } s_{\text{DYN}}^1 = \top \\ \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_2 \Rightarrow [e_{\text{DYN}}^2], s_{\text{DYN}}^2 \quad sp \text{ of } s_{\text{DYN}}^2 = \top \\ VE = \text{Comp}(FE, s) \quad C' = C + \text{Stat } VE \quad E_{\text{DYN}} = \text{Dyn}(FE + VE) \\ C', \text{exn}, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, e_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, e_{\text{DYN}}^2) \Rightarrow v \end{array}}{s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow v} \quad (225)$$

$$\frac{\begin{array}{l} \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_1 \Rightarrow v_{\text{DYN}}, s_{\text{DYN}}^1 \quad sp \text{ of } s_{\text{DYN}}^1 = \top \\ \text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_2 \Rightarrow [e_{\text{DYN}}], s_{\text{DYN}}^2 \quad sp \text{ of } s_{\text{DYN}}^2 = \top \end{array}}{s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow \text{false}} \quad (226)$$

$$\frac{\text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_1 \Rightarrow [e_{\text{DYN}}], s_{\text{DYN}}^1 \quad \text{sp of } s_{\text{DYN}}^1 = \top}{\text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_2 \Rightarrow v_{\text{DYN}}, s_{\text{DYN}}^2 \quad \text{sp of } s_{\text{DYN}}^2 = \top} \quad (227)$$

$$s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow \text{false}$$

$$\frac{s, (FE, \gamma_1) \vdash \text{exp}_1 \text{ terminates} \Rightarrow \text{false}, s}{s, (FE, \gamma_2) \vdash \text{exp}_2 \text{ terminates} \Rightarrow \text{false}, s} \quad (228)$$

$$s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow \text{true}$$

$$\frac{\text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_1 \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, s_{\text{DYN}}^1 \quad \text{sp of } s_{\text{DYN}}^1 = \top}{s, (FE, \gamma_2) \vdash \text{exp}_2 \text{ terminates} \Rightarrow \text{false}, s} \quad (229)$$

$$s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow \text{false}$$

$$\frac{s, (FE, \gamma_1) \vdash \text{exp}_1 \text{ terminates} \Rightarrow \text{false}, s}{\text{Dyn}(s, FE) \vdash_{\text{DYN}} \text{exp}_2 \Rightarrow v_{\text{DYN}}/p_{\text{DYN}}, s_{\text{DYN}}^2 \quad \text{sp of } s_{\text{DYN}}^2 = \top} \quad (230)$$

$$s, FE, (C, \tau) \cdot \gamma_1 \cdot \gamma_2 \vdash \text{exp}_1 \equiv \text{exp}_2 \Rightarrow \text{false}$$

Comments:

- (224),(225) The last sentence in the premise of both rules refers to the comparison rules (for comparing *values*), 232 to 231.
- (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 $s, M \vdash \text{exp} \text{ terminates} \Rightarrow \text{false}, s$ (see the premises of these rules) ensures that exp has no semantic value, i.e. the rules do not overlap. This is guaranteed by the rules for the convergence predicate, see rule 217.

The predicate \equiv is a partial congruence relation. It is partial because it is undefined if the evaluation of exp_1 or exp_2 involves specification constructs.

Comparisons

$$\boxed{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow v}$$

$$\frac{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{true}}{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \sqsubseteq (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \Rightarrow \text{true}} \quad (231)$$

$$C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{true}$$

$$\frac{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{false}}{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{false}} \quad (232)$$

$$\frac{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \sqsubseteq (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \Rightarrow \text{false}}{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \approx (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{false}} \quad (233)$$

$$\begin{array}{c}
\text{Approximations} \quad \boxed{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow v} \\
\\
\frac{\begin{array}{c}
id \notin \text{Dom}(VE \text{ of } C) \quad C + \{id \mapsto \tau\} \vdash_{\text{STAT}} \text{exp} \Rightarrow \{\}, \emptyset, \gamma \\
s_{\text{DYN}}^1, E_{\text{DYN}} + \{id \mapsto v_{\text{DYN}}^1\} \vdash_{\text{DYN}} \text{exp} \Rightarrow \{\}, s_{\text{DYN}} \quad \text{sp of } s_{\text{DYN}} = \top \\
\neg \exists s'_{\text{DYN}}. (s_{\text{DYN}}^2, E_{\text{DYN}} + \{id \mapsto v_{\text{DYN}}^2\} \vdash_{\text{DYN}} \text{exp} \Rightarrow \{\}, s'_{\text{DYN}} \quad \text{sp of } s'_{\text{DYN}} = \top)
\end{array}}{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{false}} \quad (234)
\end{array}$$

$$\left(\frac{\begin{array}{c}
id \notin \text{Dom}(VE \text{ of } C) \quad C + \{id \mapsto \tau\} \vdash_{\text{STAT}} \text{exp} \Rightarrow \{\}, \emptyset, \gamma \\
s_{\text{DYN}}^1, E_{\text{DYN}} + \{id \mapsto v_{\text{DYN}}^1\} \vdash_{\text{DYN}} \text{exp} \Rightarrow \{\}, s_{\text{DYN}} \quad \text{sp of } s_{\text{DYN}} = \top \\
\exists s'_{\text{DYN}}. (s_{\text{DYN}}^2, E_{\text{DYN}} + \{id \mapsto v_{\text{DYN}}^2\} \vdash_{\text{DYN}} \text{exp} \Rightarrow \{\}, s'_{\text{DYN}} \quad \text{sp of } s'_{\text{DYN}} = \top)
\end{array}}{C, \tau, E_{\text{DYN}} \vdash (s_{\text{DYN}}^1, v_{\text{DYN}}^1) \sqsubseteq (s_{\text{DYN}}^2, v_{\text{DYN}}^2) \Rightarrow \text{true}} \right) \quad (235)$$

Comments:

(234),(235) The *exp* is an arbitrary expression of type $\{\}$, the empty record type. This type contains only one value, the empty map $\{\}$. We only compare the termination behaviour of *exp*.

(224)–(235) The exception convention does not apply to these rules.

$$\text{Matches} \quad \boxed{M, v \vdash \text{match} \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{(FE, \gamma), v \vdash \text{mrule} \Rightarrow v'}{(FE, \gamma \langle \cdot \gamma' \rangle), v \vdash \text{mrule} \langle \mid \text{match} \rangle \Rightarrow v'} \quad (236)$$

$$\frac{M, v \vdash \text{mrule} \Rightarrow \text{FAIL}}{M, v \vdash \text{mrule} \Rightarrow \text{FAIL}} \quad (237)$$

$$\frac{(FE, \gamma), v \vdash \text{mrule} \Rightarrow \text{FAIL} \quad (FE, \gamma'), v \vdash \text{match} \Rightarrow v'/\text{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash \text{mrule} \mid \text{match} \Rightarrow v'/\text{FAIL}} \quad (238)$$

$$\text{Match Rules} \quad \boxed{M, v \vdash \text{mrule} \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{(FE, \gamma), v \vdash \text{pat} \Rightarrow VE \quad (FE + VE, \gamma') \vdash \text{exp} \Rightarrow v'}{(FE, \gamma \cdot \gamma'), v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow v'} \quad (239)$$

$$\frac{(FE, \gamma), v \vdash \text{pat} \Rightarrow \text{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \text{FAIL}} \quad (240)$$

Declarations

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

$$\frac{\gamma \notin \text{TraceScheme} \quad \alpha^{(k)} \cap \text{tyvars}(FE) = \emptyset \quad (FE, \gamma) \vdash \text{valbind} \Rightarrow VE \quad VE' = \text{Clos}_{FE} VE}{(FE, \forall \alpha^{(k)}. \gamma) \vdash \text{val valbind} \Rightarrow VE' \text{ in Env}} \quad (241)$$

$$\frac{s_1, (FE, \gamma) \vdash \text{typbind} \Rightarrow \{\}, s_2}{s_1, (FE, TE \cdot \gamma) \vdash \text{type typbind} \Rightarrow s_2^\#(TE) \text{ in Env, } s_2} \quad (242)$$

$$\frac{s_1, (FE, \gamma) \vdash \text{typbind} \Rightarrow \{\}, s_2}{s_1, (FE, TE \cdot \gamma) \vdash \text{eqtype typbind} \Rightarrow s_2^\#(TE) \text{ in Env, } s_2} \quad (243)$$

$$\frac{TI = \text{Fresh}(\text{Tnames}(TE \text{ of } E_{\text{STAT}}), s) \quad s_1 = s + TI \quad CT = s_1^\#(\text{Graph}(VE \text{ of } E_{\text{STAT}})) \quad s_1, (FE, \gamma) \vdash \text{datbind} \Rightarrow VE, s_1}{s, (FE, E_{\text{STAT}} \cdot \gamma) \vdash \text{datatype datbind} \Rightarrow s_1^\#(VE, TE \text{ of } E_{\text{STAT}}) \text{ in Env, } s_1 + CT} \quad (244)$$

$$\frac{TI_1 = \text{Fresh}(\text{Tnames}(TE \text{ of } E_{\text{STAT}}), s) \quad s_1 = s + TI_1 \quad CT = s_1^\#(\text{Graph}(VE \text{ of } E_{\text{STAT}})) \quad s_1, (FE, \gamma) \vdash \text{datbind} \Rightarrow VE, s_1 \quad s_1 + CT, (FE + s_1^\#(VE), \gamma') \vdash dec \Rightarrow E, s_2 \quad TI_2 = \text{Fresh}(\varphi_{\text{Ty}}^{-1}(\text{Tnames}(TE \text{ of } E_{\text{STAT}})), s_2) \quad \varphi'_{\text{Ty}} = TI_1 \circ \varphi_{\text{Ty}} \circ TI_2^{-1} \quad E = \varphi'_{\text{Ty}}(E') \quad \text{tynames}(E') \cap \text{Yield}(\varphi'_{\text{Ty}}) = \emptyset \quad s_3 = s_2 + \varphi'_{\text{Ty}} + TI_2}{s, (FE, (E_{\text{STAT}}, \varphi_{\text{Ty}}) \cdot \gamma \cdot \gamma') \vdash \text{abstype datbind with dec end} \Rightarrow E', s_3} \quad (245)$$

$$\frac{s, M \vdash \text{exbind} \Rightarrow VE, s'}{s, M \vdash \text{exception exbind} \Rightarrow s^\#(VE) \text{ in Env, } s'} \quad (246)$$

$$\frac{(FE, \gamma) \vdash dec_1 \Rightarrow E \quad (FE + E, \gamma') \vdash dec_2 \Rightarrow E'}{(FE, \gamma \cdot \gamma') \vdash \text{local } dec_1 \text{ in } dec_2 \text{ end} \Rightarrow E'} \quad (247)$$

$$\frac{M(\text{longstrid}_1) = (E_1, m_1) \quad \dots \quad M(\text{longstrid}_n) = (E_n, m_n)}{s, M \vdash \text{open longstrid}_1 \dots \text{longstrid}_n \Rightarrow E_1 + \dots + E_n, s} \quad (248)$$

$$\frac{}{M \vdash \Rightarrow \{\} \text{ in Env}} \quad (249)$$

$$\frac{(FE, \gamma) \vdash dec_1 \Rightarrow E \quad (FE + E, \gamma') \vdash dec_2 \Rightarrow E'}{(FE, \gamma \cdot \gamma') \vdash dec_1 \langle ; \rangle dec_2 \Rightarrow E + E'} \quad (250)$$

Comments:

- (241) 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.
- (244),(245) 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 verifying a *datbind*, i.e. the requirement that no state change takes place does not restrict the applicability of the rule.
- (245) The type realisation φ_{Ty} maps type names of abstract types to their implementing types, see rule 20 and the definition of Abs_{C} . Because both the type names of the abstract types and their implementing types are renamed, we have to pre- and postcompose φ_{Ty} with these renamings to obtain the corresponding realisation φ'_{Ty} that operates on the new type names. We apply it “backwards” to the environment *E* to obtain an environment *E'* in which the implementing types ($\text{Yield}(\varphi'_{\text{Ty}})$) have been replaced by the corresponding abstract type names. Such an *E'* always exists because φ'_{Ty} maps type names to type names, and it is uniquely determined because φ'_{Ty} is injective and because we require that *E'* does not contain type names of the implementing types.

Value Bindings

$$\boxed{M \vdash \text{valbind} \Rightarrow VE/p}$$

$$\frac{(FE, \gamma') \vdash \text{exp} \Rightarrow v \quad (FE, \gamma), v \vdash \text{pat} \Rightarrow VE \quad \langle (FE, \gamma'') \vdash \text{valbind} \Rightarrow VE' \rangle}{(FE, \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle) \vdash \text{pat} = \text{exp} \langle \text{and valbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (251)$$

$$\frac{(FE, \gamma') \vdash \text{exp} \Rightarrow v \quad (FE, \gamma), v \vdash \text{pat} \Rightarrow \text{FAIL}}{(FE, \gamma \cdot \gamma' \langle \cdot \gamma'' \rangle) \vdash \text{pat} = \text{exp} \langle \text{and valbind} \rangle \Rightarrow [\text{Bind}]} \quad (252)$$

$$\frac{M \vdash \text{valbind} \Rightarrow VE}{M \vdash \text{rec valbind} \Rightarrow \text{Rec } VE} \quad (253)$$

Comments:

- (251) 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

$$\boxed{M \vdash \text{typbind} \Rightarrow \{\}} \quad (254)$$

$$\frac{\langle (FE, \gamma) \vdash \text{typbind} \Rightarrow \{\} \rangle}{(FE, \epsilon \langle \cdot \gamma \rangle) \vdash \text{tyvarseq tycon} = \text{ty} \langle \text{and typbind} \rangle \Rightarrow \{\}} \quad (254)$$

$$\frac{\begin{array}{l} n = n \text{ of } s \quad (QIT \text{ of } FE)(n) = \text{ty} \quad s^\#(C) \vdash_{\text{STAT}} \text{ty} \Rightarrow \tau \\ \text{tyvarseq} = \alpha^{(k)} \quad \text{tyvars}(\text{ty}) \subseteq \alpha^{(k)} \quad TI = \{t \mapsto \Lambda \alpha^{(k)}. \tau\} \\ s' = (s + TI) \oplus (n + 1) \quad \langle s', (FE, \gamma) \vdash \text{typbind} \Rightarrow \{\}, s'' \rangle \end{array}}{s, (FE, (C, t) \langle \cdot \gamma \rangle) \vdash \text{tyvarseq tycon} = ? \langle \text{and typbind} \rangle \Rightarrow \{\}, s' \langle ' \rangle} \quad (255)$$

Comments:

(255) 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

$$\boxed{M \vdash \text{datbind} \Rightarrow VE} \quad (256)$$

$$\frac{(FE, \gamma) \vdash \text{conbind} \Rightarrow VE \quad \langle (FE, \gamma') \vdash \text{datbind} \Rightarrow VE' \rangle}{(FE, \gamma \langle \cdot \gamma' \rangle) \vdash \text{tyvarseq tycon} = \text{conbind} \langle \text{and datbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (256)$$

Constructor Bindings

$$\boxed{M \vdash \text{conbind} \Rightarrow VE} \quad (257)$$

$$\frac{\text{con} = \text{id}^c \quad \langle (FE, \gamma) \vdash \text{conbind} \Rightarrow VE \rangle}{(FE, \tau \langle \cdot \gamma \rangle) \vdash \text{con} \langle | \text{conbind} \rangle \Rightarrow \{ \text{id} \mapsto \text{Clos}(\text{con}, \tau) \} \langle + VE \rangle} \quad (257)$$

$$\frac{\text{con} = \text{id}^c \quad \langle (FE, \gamma) \vdash \text{conbind} \Rightarrow VE \rangle}{(FE, \tau \langle \cdot \gamma \rangle) \vdash \text{con of ty} \langle | \text{conbind} \rangle \Rightarrow \{ \text{id} \mapsto \text{Clos}(\text{con}, \tau) \} \langle + VE \rangle} \quad (258)$$

Exception Bindings

$$\boxed{M \vdash \text{exbind} \Rightarrow VE} \quad (259)$$

$$\frac{\begin{array}{l} \text{excon} = \text{id}^e \quad \text{en} = \min(\text{ExName} \setminus (\text{ens of } s)) \\ s' = s + (\{ \text{en} \} \text{ in ExNameSet}, \{ (\text{en}, \text{exn}) \} \text{ in ValTemp}) \\ \langle s', (FE, \gamma) \vdash \text{exbind} \Rightarrow VE, s'' \rangle \end{array}}{s, (FE, \epsilon \langle \cdot \gamma \rangle) \vdash \text{excon} \langle \text{and exbind} \rangle \Rightarrow \{ \text{id} \mapsto (\text{en}, \text{exn}) \} \langle + VE \rangle, s' \langle ' \rangle} \quad (259)$$

$$\frac{\begin{array}{l} \text{excon} = \text{id}^e \quad \text{en} = \min(\text{ExName} \setminus (\text{ens of } s)) \quad \tau' = s^\#(\tau) \rightarrow \text{exn} \\ s' = s + (\{ \text{en} \} \text{ in ExNameSet}, \{ (\text{en}, \tau') \} \text{ in ValTemp}) \\ \langle s', (FE, \gamma) \vdash \text{exbind} \Rightarrow VE, s'' \rangle \end{array}}{s, (FE, \tau \langle \cdot \gamma \rangle) \vdash \text{excon of ty} \langle \text{and exbind} \rangle \Rightarrow \{ \text{id} \mapsto (\text{en}, \tau') \} \langle + VE \rangle, s' \langle ' \rangle} \quad (260)$$

$$\frac{excon = id^e \quad FE(longexcon) = (en, \tau) \quad \langle (FE, \gamma) \vdash exbind \Rightarrow VE \rangle}{(FE, \epsilon \langle \cdot \gamma \rangle) \vdash excon = longexcon \langle \text{and } exbind \rangle \Rightarrow \{id \mapsto (en, \tau)\} \langle + VE \rangle} \quad (261)$$

Comments:

(259),(260) The use of min ensures that *excon* is uniquely determined.

Atomic Patterns

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

$$\overline{M, v \vdash - \Rightarrow \{}} \quad (262)$$

$$\frac{v = \text{val}(scon)}{M, v \vdash scon \Rightarrow \{}} \quad (263)$$

$$\frac{v \neq \text{val}(scon)}{M, v \vdash scon \Rightarrow \text{FAIL}} \quad (264)$$

$$\frac{var = id^V}{s, (FE, \tau), v \vdash var \Rightarrow \{id \mapsto (v, s^\#(\tau))\}, s} \quad (265)$$

$$\frac{M(longcon) \succ (con, \tau) \quad v = con}{M, v \vdash longcon \Rightarrow \{}} \quad (266)$$

$$\frac{M(longcon) \succ (con, \tau) \quad v \neq con}{M, v \vdash longcon \Rightarrow \text{FAIL}} \quad (267)$$

$$\frac{M(longexcon) = (v, \tau)}{M, v \vdash longexcon \Rightarrow \{}} \quad (268)$$

$$\frac{M(longexcon) = (v', \tau) \quad v' \neq v}{M, v \vdash longexcon \Rightarrow \text{FAIL}} \quad (269)$$

$$\frac{v = \{ \} \langle +r \rangle \text{ in Val} \quad \langle (FE, \gamma), r \vdash patrow \Rightarrow VE/FAIL \rangle}{(FE, \epsilon \langle \cdot \gamma \rangle), v \vdash \{ \langle patrow \rangle \} \Rightarrow \{ \} \langle +VE/FAIL \rangle} \quad (270)$$

$$\frac{M, v \vdash pat \Rightarrow VE/FAIL}{M, v \vdash (pat) \Rightarrow VE/FAIL} \quad (271)$$

Pattern Rows

$$\boxed{M, r \vdash \text{patrow} \Rightarrow VE/\text{FAIL}}$$

$$\overline{M, r \vdash \dots \Rightarrow \{\}} \quad (272)$$

$$\frac{(FE, \gamma), r(\text{lab}) \vdash \text{pat} \Rightarrow \text{FAIL}}{(FE, \gamma \langle \cdot \gamma' \rangle), r \vdash \text{lab} = \text{pat} \langle \ , \text{patrow} \rangle \Rightarrow \text{FAIL}} \quad (273)$$

$$\frac{(FE, \gamma), r(\text{lab}) \vdash \text{pat} \Rightarrow VE \quad \langle (FE, \gamma'), r \vdash \text{patrow} \Rightarrow VE'/\text{FAIL} \rangle}{(FE, \gamma \langle \cdot \gamma' \rangle), r \vdash \text{lab} = \text{pat} \langle \ , \text{patrow} \rangle \Rightarrow VE \langle + VE'/\text{FAIL} \rangle} \quad (274)$$

Patterns

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

$$\frac{M, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{M, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}} \quad (275)$$

$$\frac{FE(\text{longcon}) \succ (con, \tau') \quad v = (con, v') \quad (FE, \gamma), v' \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{(FE, \tau \cdot \gamma), v \vdash \text{longcon atpat} \Rightarrow VE/\text{FAIL}} \quad (276)$$

$$\frac{M(\text{longcon}) \succ (con, \tau) \quad v \notin \{con\} \times \text{Val}}{M, v \vdash \text{longcon atpat} \Rightarrow \text{FAIL}} \quad (277)$$

$$\frac{M(\text{longexcon}) = (en, \tau) \quad v = (en, v') \quad M, v' \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{M, v \vdash \text{longexcon atpat} \Rightarrow VE/\text{FAIL}} \quad (278)$$

$$\frac{M(\text{longexcon}) = (en, \tau) \quad v \notin \{en\} \times \text{Val}}{M, v \vdash \text{longexcon atpat} \Rightarrow \text{FAIL}} \quad (279)$$

$$\frac{M, v \vdash \text{pat} \Rightarrow VE/\text{FAIL}}{M, v \vdash \text{pat} : \text{ty} \Rightarrow VE/\text{FAIL}} \quad (280)$$

$$\frac{(FE, \gamma), v \vdash \text{var} \Rightarrow VE \quad (FE, \gamma'), v \vdash \text{pat} \Rightarrow VE'/\text{FAIL}}{(FE, \gamma \cdot \gamma'), v \vdash \text{var} \langle : \text{ty} \rangle \text{ as pat} \Rightarrow VE + VE'/\text{FAIL}} \quad (281)$$

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 255).

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 $\text{Set}(A) = \wp(\text{State} \times A)$; each $\mathcal{A} \in \text{Set}(A)$ is a (possibly infinite) set of pairs (s, a) , $s \in \text{State}$, $a \in A$. We use the same conventions as in the verification semantics for the Core to refer to semantic objects of the static and dynamic semantics.

$$\begin{aligned}
(B, \gamma, axdesc) &\in \text{BasicAx} = \text{Basis} \times \text{Trace} \times \text{AxDesc} \\
(N, B)(I, A) &\in \text{ExistAx} = \text{NameSet} \times \text{Basis} \times (\text{Int} \times \text{GenAx}) \\
A &\in \text{GenAx} = \text{BasicAx} \uplus \text{ExistAx} \uplus \\
&\quad (\text{GenAx} \times \text{GenAx}) \uplus \text{Bit} \\
(IE, TE, VE_{\text{STAT}}, A) \text{ or } I &\in \text{Int} = \text{IntEnv} \times \text{TyEnv} \times \\
&\quad \text{VarEnv}_{\text{STAT}} \times \text{GenAx} \\
(I, m) \text{ or } IS &\in \text{IntStr} = \text{Int} \times \text{StrName} \\
IE &\in \text{IntEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{IntStr} \\
(N)IS \text{ or } \Sigma &\in \text{Sig} = \text{NameSet} \times \text{IntStr} \\
ostrexp &\in \text{OptStrExp} = \text{StrExp} \uplus \{?\} \\
(strid, ostrexp, B, (N)(IS, \Sigma)) &\in \text{FunctorClosure} = \\
&\quad \text{StrId} \times \text{OptStrExp} \times \\
&\quad \text{Basis} \times \text{NameSet} \times \text{IntStr} \times \text{Sig} \\
G &\in \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\
F &\in \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunctorClosure} \\
(N, F, G, E) \text{ or } B &\in \text{Basis} = \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
(N, F, G, IE, E) \text{ or } IB &\in \text{IntBasis} = \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \\
&\quad \text{IntEnv} \times \text{Env} \\
\mathcal{S} &\in \text{Set}(\text{Str}) \\
\mathcal{R} &\in \text{Set}(\text{Rea} \times \text{Str}) \\
\mathcal{E} &\in \text{Set}(\text{Env}) \\
\mathcal{SE} &\in \text{Set}(\text{StrEnv}) \\
\mathcal{B} &\in \text{Set}(\text{Basis})
\end{aligned}$$

Figure 18: Further Compound Semantic Objects

An *interface* $I \in \text{Int}$ 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* $\Sigma \in \text{Sig}$ 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* $(strid, ostrexp, B, (N)(IS, \Sigma))$ has four 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, the rôle of the functor body $ostrexp$ is less significant, because the semantics of a functor only depends on its interface, not on its implementation. The functor body $ostrexp$ is only of significance for the generation of “arbitrary” structures. The last 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 $\text{Inter} : \text{Env} \rightarrow \text{Int}$ as follows:

$$\begin{aligned} \text{Inter}(SE, TE, VE) &= (IE, TE, \text{Stat } VE, \perp) \\ \text{where } IE &= \{strid \mapsto (\text{Inter } E, m) ; SE(strid) = (E, m)\} \end{aligned}$$

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 verifying 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:

$$\begin{aligned} \text{Dom } SE' &= \text{Dom } SE \setminus \text{Dom}(IE \text{ of } I) \\ \text{Dom } TE' &= \text{Dom } TE \setminus \text{Dom}(TE \text{ of } I) \\ \text{Dom } VE' &= \text{Dom } VE \setminus \text{Dom}(VE_{\text{STAT}} \text{ of } I) \end{aligned}$$

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 verifying 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, $(VE_{\text{STAT}}$ in Int) stands for $(\{\}, \{\}, VE_{\text{STAT}}, \top)$. The generalised axiom \perp is never satisfied.

9.3 Combining Interfaces

Given two interfaces I_1 and I_2 with $I_k = (IE_k, TE_k, VE_{\text{STAT}}^k, A_k)$, we define their sequential composition $I_1 \oplus I_2$ as follows:

$$I_1 \oplus I_2 = (IE_1 + IE_2, TE_1 + TE_2, VE_{\text{STAT}}^1 + VE_{\text{STAT}}^2, (A_1, A_2))$$

Notice that the restrictions imposed by the static semantics make sure that the combined maps have always disjoint domains.

9.4 Extracting Objects of the Static Semantics

We extend the family of functions Stat defined in Section 8.7 to semantic objects of the module semantics as follows:

$$\begin{aligned} \text{Stat}(N, F, G, E) &= (N, \text{Stat } F, \text{Stat } \circ G, \text{Stat } E) \\ \text{Stat}(N, F, G, IE, E) &= (N, \text{Stat } F, \text{Stat } \circ G, \text{Stat } E + (\text{Stat } \circ IE)) \\ \text{Stat } F &= \{funid \mapsto (N)(\text{Stat } IS, \text{Stat } \Sigma) ; \\ &\quad F(funid) = (strid, B, (N)(IS, \Sigma))\} \\ \text{Stat}((N)IS) &= (N) \text{Stat } IS \\ \text{Stat}(I, m) &= (m, \text{Stat } I) \\ \text{Stat}(IE, TE, VE_{\text{STAT}}, A) &= (\text{Stat } \circ IE, TE, VE_{\text{STAT}}) \end{aligned}$$

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 304.

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

$$\frac{\text{Supp } \varphi \subseteq N \quad \varphi(IS) = (I, m) \quad s, E \vdash I \Rightarrow E'}{s, (N)IS \vdash (E, m) \Rightarrow (E', m), \varphi} \quad (282)$$

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.

$$\frac{s, SE \vdash IE \Rightarrow SE' \quad TE \vdash TE' \Rightarrow TE'' \quad VE \vdash VE_{\text{STAT}} \Rightarrow VE' \quad E' = (SE', TE'', VE') \quad s, E' \vdash A \Rightarrow \{\}}{s, (SE, TE, VE) \vdash (IE, TE', VE_{\text{STAT}}, A) \Rightarrow E'} \quad (283)$$

Restricting Structure Environments

$$\boxed{s, SE \vdash IE \Rightarrow SE'}$$

$$\frac{s, SE \vdash IE \Rightarrow SE' \quad s, E \vdash I \Rightarrow E'}{s, SE + \{\text{strid} \mapsto (E, m)\} \vdash IE + \{\text{strid} \mapsto (I, m)\} \Rightarrow SE' + \{\text{strid} \mapsto (E', m)\}} \quad (284)$$

$$\overline{s, SE \vdash \{\} \Rightarrow \{\}} \quad (285)$$

Restricting Type Environments

$$\boxed{TE \vdash TE' \Rightarrow TE''}$$

$$\frac{TE \vdash TE' \Rightarrow TE'' \quad CE = CE' \vee CE' = \{\}}{TE + \{\text{tycon} \mapsto (\theta, CE)\} \vdash TE' + \{\text{tycon} \mapsto (\theta, CE')\} \Rightarrow TE'' + \{\text{tycon} \mapsto (\theta, CE')\}} \quad (286)$$

$$\overline{TE \vdash \{\} \Rightarrow \{\}} \quad (287)$$

Comments:

(286) Notice that the side-condition closely relates to the enrichment of type structures, see section 5.11.

Restricting Variable Environments

$$\boxed{VE \vdash VE_{\text{STAT}} \Rightarrow VE'}$$

$$\frac{VE \vdash VE_{\text{STAT}} \Rightarrow VE' \quad \forall \alpha^{(k)}. \tau \succ_{\vartheta} \forall \beta^{(l)}. \tau'}{VE + \{\text{id} \mapsto \forall \alpha^{(k)}. (v, \tau)\} \vdash VE_{\text{STAT}} + \{\text{id} \mapsto \forall \beta^{(l)}. \tau'\} \Rightarrow VE' + \{\text{id} \mapsto \forall \beta^{(l)}. (\vartheta(v), \tau')\}} \quad (288)$$

$$\overline{VE \vdash \{\} \Rightarrow \{\}} \quad (289)$$

Comments:

(288) There is a certain amount of arbitrariness in this rule, similarly as in rules 195 and 241 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

$$\boxed{s, E \vdash A \Rightarrow \{\}}$$

$$\frac{l \text{ of } s = \perp}{s, E \vdash A \Rightarrow \{\}} \quad (290)$$

$$\frac{}{s, E \vdash \top \Rightarrow \{\}} \quad (291)$$

$$\frac{s, E \vdash A_1 \Rightarrow \{\} \quad s, E \vdash A_2 \Rightarrow \{\}}{s, E \vdash (A_1, A_2) \Rightarrow \{\}} \quad (292)$$

$$\frac{l \text{ of } s = \top \quad s, B + E, \gamma \vdash \text{axdesc} \Rightarrow \{\}}{s, E \vdash (B, \gamma, \text{axdesc}) \Rightarrow \{\}} \quad (293)$$

$$\frac{m \notin (N \text{ of } B \cup N) \quad l \text{ of } s = \top \quad s_{\perp}, B, (N)(I, m) \vdash \text{strex}^{\bullet} \Rightarrow \mathcal{R} \quad (s', (\varphi, (E', m'))) \in \mathcal{R} \quad s'_{\top}, E + E' \vdash \varphi(A) \Rightarrow \{\}}{s, E \vdash (N, B)(I, A) \Rightarrow \{\}} \quad (294)$$

Comments:

(290) Axioms in signatures are ignored if the flag l of the state is \perp .

(294) 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.

Validating Axiom Descriptions

$$\boxed{s, B, \gamma \vdash \text{axdesc} \Rightarrow \{\}}$$

$$\frac{s, B, \gamma_1 \vdash \text{specexp} \Rightarrow \{\} \quad \langle s, B, \gamma_2 \vdash \text{axdesc} \Rightarrow \{\} \rangle}{s, B, \gamma_1 \langle \cdot \gamma_2 \rangle \vdash \text{specexp} \langle \text{and axdesc} \rangle \Rightarrow \{\}} \quad (295)$$

Validating Specification Expressions

$$\boxed{s, B, \gamma \vdash \text{specexp} \Rightarrow \{\}}$$

$$\frac{s, B, \gamma_1 \vdash \text{strdec} \Rightarrow \mathcal{E} \quad \frac{(s', E) \in \mathcal{E}}{s', B + E, \gamma_2 \vdash \text{axexp} \Rightarrow \{\}}}{s, B, \gamma_1 \cdot \gamma_2 \vdash \text{let strdec in axexp end} \Rightarrow \{\}} \quad (296)$$

Axiomatic Expressions

$$\boxed{s, B, \gamma \vdash \text{axexp} \Rightarrow \{\}}$$

$$\frac{\left(\frac{\gamma \succ \gamma'}{s, ((E \text{ of } B, (\{\}, \{\})), \gamma') \vdash \text{exp}^{\bullet} \Rightarrow \text{true}, s'} \right)}{s, B, \gamma \vdash \text{exp}^{\bullet} \Rightarrow \{\}} \quad (297)$$

Comment: An axiomatic expression exp^{\bullet} holds if it verifies 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

$$\boxed{s, B, \gamma \vdash \text{strex} \Rightarrow \mathcal{S}}$$

$$\frac{s, B, \gamma \vdash \text{strdec} \Rightarrow \mathcal{E}}{s, B, m \cdot \gamma \vdash \mathbf{struct} \text{ strdec} \mathbf{end} \Rightarrow \{(s', (E, m)) \mid (s', E) \in \mathcal{E}\}} \quad (298)$$

$$\frac{B(\text{longstrid}) = S}{s, B, \gamma \vdash \text{longstrid} \Rightarrow \{(s, S)\}} \quad (299)$$

$$\frac{\begin{array}{l} B(\text{funid}) = (\text{strid}, \text{ostrex}, B', (N)(IS, \Sigma)) \quad \varphi' = s^\#(\varphi) \\ l \text{ of } s = \top \quad \varphi'(IS, \Sigma) = (IS', \Sigma') \quad s, B, \gamma \vdash \text{strex} \Rightarrow \mathcal{S} \\ TI = \text{Fresh}(N, s) \quad \frac{(s', S) \in \mathcal{S}}{\exists S', \varphi''. s', (N)IS' \vdash S \Rightarrow S', \varphi''} \end{array}}{s, B, \varphi \cdot \gamma \vdash \text{funid} (\text{strex}) \Rightarrow \{(s'_\top, S') \mid (s', S) \in \mathcal{S}, \\ s'_\perp, B' \oplus \{\text{strid} \mapsto S\}, TI(\Sigma') \vdash \text{strex}^\bullet \Rightarrow \mathcal{R}, \\ (s'', (\varphi'', S')) \in \mathcal{R}\}} \quad (300)$$

$$\frac{\begin{array}{l} B(\text{funid}) = (\text{strid}, \text{strex}', B', (N)(IS, \Sigma)) \quad \varphi' = s^\#(\varphi) \\ l \text{ of } s = \perp \quad \varphi'(IS, \Sigma) = (IS', \Sigma') \quad s, B, \gamma \vdash \text{strex} \Rightarrow \mathcal{S} \\ TI = \text{Fresh}(N, s) \quad \frac{(s', S) \in \mathcal{S}}{\exists S', \varphi''. s', (N)IS' \vdash S \Rightarrow S', \varphi''} \end{array}}{s, B, \varphi \cdot \gamma \vdash \text{funid} (\text{strex}') \Rightarrow \{(s'', S') \mid (s', S) \in \mathcal{S}, \\ s', B' \oplus \{\text{strid} \mapsto S\}, TI(\Sigma') \vdash \text{strex}' \Rightarrow \mathcal{R}, \\ (s'', (\varphi'', S')) \in \mathcal{R}\}} \quad (301)$$

$$\frac{s, B, \gamma \vdash \text{strdec} \Rightarrow \mathcal{E} \quad \frac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{S}. s_1, B \oplus E, \gamma' \vdash \text{strex} \Rightarrow \mathcal{S}}}{s, B, \gamma \cdot \gamma' \vdash \mathbf{let} \text{ strdec} \mathbf{in} \text{strex} \mathbf{end} \Rightarrow \{(s_2, S) \mid (s_1, E) \in \mathcal{E}, s_1, B \oplus E, \gamma' \vdash \text{strex} \Rightarrow \mathcal{S}, (s_2, S) \in \mathcal{S}\}} \quad (302)$$

Comments:

(300) It should be emphasised here that *strex* and *strex*[•] are different meta-variables; *strex*[•] is an *arbitrary* structure expression (not containing ?). There are no explicit restrictions concerning elaboration of *strex*[•] needed, because they are implicit in the sentence form used, see rule 303.

(301) For this rule to succeed, it is necessary that the *ostrex* component of *B(funid)* is a proper structure expression and not a ? — this is implicit in the first equation.

Structure vs. Signature

$$\boxed{s, B, \Sigma \vdash \text{strex} \Rightarrow \mathcal{R}}$$

Sentences of this form can be read as: in state s and basis B , the structure expression strex verifies 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 strex to Σ .

$$\frac{\text{Stat } B \vdash_{\text{STAT}} \text{strex} \Rightarrow (m, E_{\text{STAT}}), \gamma \quad \frac{(s_2, S) \in \mathcal{S}}{\exists S', \varphi. s_2, \Sigma \vdash S \Rightarrow S', \varphi}}{s_1, B, \gamma \vdash \text{strex} \Rightarrow \mathcal{S}} \quad (303)$$

$$s_1, B, \Sigma \vdash \text{strex} \Rightarrow \{(s_2 + \varphi_{\text{Ty}}, (\varphi, S')) \mid (s_2, S) \in \mathcal{S}, s_2, \Sigma \vdash S \Rightarrow S', \varphi\}$$

Comment: $s_2 + \varphi_{\text{Ty}}$ 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

$$\boxed{s, B, \gamma \vdash \text{strdec} \Rightarrow \mathcal{E}}$$

Notice that the semantic value of a structure declaration is an \mathcal{E} , not an \mathcal{E}/p .

$$\overline{s, (N, F, G, E), \gamma \vdash \text{dec} \Rightarrow \{(s', E') \mid s, (E, QI), \gamma \vdash \text{dec} \Rightarrow E', s'\}} \quad (304)$$

$$\overline{s, B, \gamma \vdash \text{axiom } ax \Rightarrow \{(s, \{\} \text{ in Env}) \mid s, B, \gamma \vdash ax \Rightarrow \{\}\}} \quad (305)$$

$$\frac{s, B, \gamma \vdash \text{strbind} \Rightarrow \mathcal{SE}}{s, B, \gamma \vdash \text{structure } \text{strbind} \Rightarrow \{(s', SE \text{ in Env}) \mid (s', SE) \in \mathcal{SE}\}} \quad (306)$$

$$\frac{s, B, \gamma_1 \vdash \text{strdec}_1 \Rightarrow \mathcal{E} \quad \frac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{E}'. s_1, B \oplus E, \gamma_2 \vdash \text{strdec}_2 \Rightarrow \mathcal{E}'}}{s, B, \gamma_1 \cdot \gamma_2 \vdash \text{local } \text{strdec}_1 \text{ in } \text{strdec}_2 \text{ end} \Rightarrow \{(s_2, E') \mid (s_1, E) \in \mathcal{E}, s_1, B \oplus E, \gamma_2 \vdash \text{strdec}_2 \Rightarrow \mathcal{E}', (s_2, E') \in \mathcal{E}'\}} \quad (307)$$

$$\overline{s, B, \gamma \vdash \Rightarrow \{(s, \{\} \text{ in Env})\}} \quad (308)$$

$$\frac{s, B, \gamma_1 \vdash \text{strdec}_1 \Rightarrow \mathcal{E} \quad \frac{(s_1, E) \in \mathcal{E}}{\exists \mathcal{E}'. s_1, B \oplus E, \gamma_2 \vdash \text{strdec}_2 \Rightarrow \mathcal{E}'}}{s, B, \gamma_1 \cdot \gamma_2 \vdash \text{strdec}_1 \langle ; \rangle \text{strdec}_2 \Rightarrow \{(s_2, E + E') \mid (s_1, E) \in \mathcal{E}, s_1, B \oplus E, \gamma_2 \vdash \text{strdec}_2 \Rightarrow \mathcal{E}', (s_2, E') \in \mathcal{E}'\}} \quad (309)$$

Comments:

(304) 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*) verifies, but it may verify to an empty set; it may also verify to a set with more than one element, because *QI* can be chosen arbitrarily. All Core verifications resulting in packets are ignored.

(305) 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

$$\boxed{s, B, \gamma \vdash ax \Rightarrow \{\}} \quad (306)$$

$$\frac{s, B, \gamma \vdash axexp \Rightarrow \{\} \quad \langle s, B, \gamma' \vdash ax \Rightarrow \{\} \rangle}{s, B, \gamma \langle \cdot \gamma' \rangle \vdash axexp \langle \text{and } ax \rangle \Rightarrow \{\}} \quad (310)$$

Structure Bindings

$$\boxed{s, B, \gamma \vdash strbind \Rightarrow \mathcal{SE}} \quad (307)$$

$$\frac{s, B, \gamma \vdash sglstrbind \Rightarrow \mathcal{SE} \quad \left\langle \frac{(s', SE) \in \mathcal{SE}}{\exists SE'. s', B + \text{names } SE, \gamma' \vdash strbind \Rightarrow SE'} \right\rangle}{s, B, \gamma \langle \cdot \gamma' \rangle \vdash sglstrbind \langle \text{and } strbind \rangle \Rightarrow \{(s' \langle \cdot \rangle, SE \langle + SE' \rangle) \mid (s', SE) \in \mathcal{SE} \langle \cdot \rangle, s', B + \text{names } SE, \gamma' \vdash strbind \Rightarrow SE', (s'', SE') \in \mathcal{SE}'\}} \quad (311)$$

Single Structure Bindings

$$\boxed{s, B, \gamma \vdash sglstrbind \Rightarrow \mathcal{SE}} \quad (308)$$

$$\frac{s, \text{Inter } B, \gamma \vdash psigexp \Rightarrow \Sigma \quad s, B, \Sigma \vdash strexp \Rightarrow \mathcal{R} \quad l \text{ of } s = \top}{s, B, \gamma \cdot \gamma' \vdash strid : psigexp = strexp \Rightarrow \{(s'_\top, \{strid \mapsto S'\}) \mid s_\perp, B, \Sigma \vdash strexp^\bullet \Rightarrow \mathcal{R}', (s', (\varphi, S')) \in \mathcal{R}'\}} \quad (312)$$

$$\frac{s, \text{Inter } B, \gamma \vdash psigexp \Rightarrow \Sigma \quad s, B, \Sigma \vdash strexp \Rightarrow \mathcal{R} \quad l \text{ of } s = \perp}{s, B, \gamma \cdot \gamma' \vdash strid : psigexp = strexp \Rightarrow \{(s', \{strid \mapsto S'\}) \mid s, B, \Sigma \vdash strexp \Rightarrow \mathcal{R}', (s', (\varphi, S')) \in \mathcal{R}'\}} \quad (313)$$

$$\frac{s, \text{Inter } B, \gamma \vdash psigexp \Rightarrow \Sigma}{s, B, \gamma \vdash strid : psigexp = ? \Rightarrow \{(s'_\top, \{strid \mapsto S'\}) \mid s_\perp, B, \Sigma \vdash strexp^\bullet \Rightarrow \mathcal{R}, (s', (\varphi, S)) \in \mathcal{R}\}} \quad (314)$$

$$\frac{s, B, \gamma \vdash strexp \Rightarrow \mathcal{S}}{s, B, \gamma \vdash strid = strexp \Rightarrow \{(s', \{strid \mapsto S'\}) \mid (s', S) \in \mathcal{S}\}} \quad (315)$$

Comments:

(312) The second premise implicitly requires *every* interpretation of *strexp* to match the signature Σ (see rule 303). The resulting set \mathcal{R} is then ignored and the result is the union of all expressible sets of structures (since *strexp*[•] 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 $s, B, \Sigma \vdash \textit{strexp} \Rightarrow \mathcal{R}$ are also used for cases in which there is no trace readily available, e.g. rules 314 and 300.

(312),(313) Notice the subtle difference between both rules: rule 313 uses *strexp* instead of *strexp*[•] in its conclusion.

Signature Expressions

$$\boxed{s, IB, \gamma \vdash \textit{sigexp} \Rightarrow IS}$$

$$\frac{s, IB, \gamma \vdash \textit{spec} \Rightarrow I}{s, IB, m \cdot \gamma \vdash \mathbf{sig\ spec\ end} \Rightarrow (I, m)} \quad (316)$$

$$\frac{IB(\textit{sigid}) \geq_{\varphi} IS}{s, IB, \varphi \vdash \textit{sigid} \Rightarrow IS} \quad (317)$$

Principal Signatures

$$\boxed{s, IB, \gamma \vdash \textit{psigexp} \Rightarrow \Sigma}$$

$$\frac{N \cap N \text{ of } IB = \emptyset \quad s, IB, \gamma \vdash \textit{sigexp} \Rightarrow IS}{s, IB, (N)\gamma \vdash \textit{sigexp} \Rightarrow (N)IS} \quad (318)$$

Comments:

(318) 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 $s, IB, \gamma \vdash \textit{sigexp} \Rightarrow IS$ uses the same realisations as the corresponding derivation of $B_{\text{STAT}} \vdash_{\text{STAT}} \textit{sigexp} \Rightarrow S_{\text{STAT}}, \gamma$ when determining the principal signature for rule 65.

Signature Declarations

$$\boxed{s, IB, \gamma \vdash \textit{sigdec} \Rightarrow G}$$

$$\frac{s, IB, \gamma \vdash \textit{sigbind} \Rightarrow G}{s, IB, \gamma \vdash \mathbf{signature\ sigbind} \Rightarrow G} \quad (319)$$

$$\frac{}{s, IB, \gamma \vdash \quad \Rightarrow \{\}} \quad (320)$$

$$\frac{s, IB, \gamma_1 \vdash \textit{sigdec}_1 \Rightarrow G_1 \quad s, IB + G_1, \gamma_2 \vdash \textit{sigdec}_2 \Rightarrow G_2}{s, IB, \gamma_1 \cdot \gamma_2 \vdash \textit{sigdec}_1 \langle ; \rangle \textit{sigdec}_2 \Rightarrow G_1 + G_2} \quad (321)$$

Signature Bindings

$$\boxed{s, IB, \gamma \vdash \text{sigbind} \Rightarrow G}$$

$$\frac{s, IB, \gamma \vdash \text{psigexp} \Rightarrow \Sigma \quad \langle s, IB, \gamma' \vdash \text{sigbind} \Rightarrow G \rangle}{s, IB, \gamma \langle \cdot \gamma' \rangle \vdash \text{sigid} = \text{psigexp} \langle \text{and sigbind} \rangle \Rightarrow \{ \text{sigid} \mapsto \Sigma \} \langle + G \rangle} \quad (322)$$

Specifications

$$\boxed{s, IB, \gamma \vdash \text{spec} \Rightarrow I}$$

$$\frac{}{s, IB, VE_{\text{STAT}} \vdash \text{val} \text{ valdesc} \Rightarrow s^\#(VE_{\text{STAT}}) \text{ in Int}} \quad (323)$$

$$\frac{}{s, IB, TE \vdash \text{type} \text{ typedesc} \Rightarrow s^\#(TE) \text{ in Int}} \quad (324)$$

$$\frac{}{s, IB, TE \vdash \text{eqtype} \text{ typedesc} \Rightarrow s^\#(TE) \text{ in Int}} \quad (325)$$

$$\frac{}{s, IB, (TE, VE_{\text{STAT}}) \text{ in Env}_{\text{STAT}} \vdash \text{datatype} \text{ datdesc} \Rightarrow (s^\#(TE), s^\#(VE_{\text{STAT}})) \text{ in Int}} \quad (326)$$

$$\frac{}{s, IB, VE_{\text{STAT}} \text{ in Env}_{\text{STAT}} \vdash \text{exception} \text{ exdesc} \Rightarrow (s^\#(VE_{\text{STAT}})) \text{ in Int}} \quad (327)$$

$$\frac{}{s, IB, \gamma \vdash \text{axiom} \text{ axdesc} \Rightarrow (B \text{ of } IB, \gamma, \text{axdesc}) \text{ in Int}} \quad (328)$$

$$\frac{s, IB, \gamma \vdash \text{strdesc} \Rightarrow IE}{s, IB, \gamma \vdash \text{structure} \text{ strdesc} \Rightarrow IE \text{ in Int}} \quad (329)$$

$$\frac{}{s, IB, \gamma \vdash \text{sharing} \text{ shareq} \Rightarrow \{ \} \text{ in Int}} \quad (330)$$

$$\frac{\begin{array}{l} IB = (N, F, G, IE, E) \quad s, IB, \gamma_1 \vdash \text{spec}_1 \Rightarrow I_1 \\ IB' = (N \cup \text{names } I_1, F, G, IE + IE \text{ of } I_1, E \setminus I_1) \quad s, IB', \gamma_2 \vdash \text{spec}_2 \Rightarrow I_2 \\ I_2 = (IE, TE, VE_{\text{STAT}}, A) \quad N_1 = \text{names } I_2 \setminus \text{names } I_1 \setminus N \\ I_3 = (IE, TE, VE_{\text{STAT}}, (N_1, B \text{ of } IB)(I_1, A)) \end{array}}{s, IB, \gamma_1 \cdot \gamma_2 \vdash \text{local} \text{ spec}_1 \text{ in spec}_2 \text{ end} \Rightarrow I_3} \quad (331)$$

$$\frac{IB(\text{longstrid}_1) = (I_1, m_1) \cdots IB(\text{longstrid}_n) = (I_n, m_n)}{s, IB, \gamma \vdash \text{open} \text{ longstrid}_1 \cdots \text{longstrid}_n \Rightarrow I_1 \oplus \cdots \oplus I_n} \quad (332)$$

$$\frac{IB(\text{sigid}_1) \geq_{\varphi_1} (I_1, m_1) \cdots IB(\text{sigid}_n) \geq_{\varphi_n} (I_n, m_n)}{s, IB, \varphi_1 \cdots \varphi_n \vdash \text{include} \text{ sigid}_1 \cdots \text{sigid}_n \Rightarrow I_1 \oplus \cdots \oplus I_n} \quad (333)$$

$$\frac{}{s, IB, \gamma \vdash \Rightarrow \{ \} \text{ in Int}} \quad (334)$$

$$\frac{IB = (N, F, G, IE, E) \quad s, IB, \gamma_1 \vdash spec_1 \Rightarrow I_1 \quad IB' = (N \cup \text{names } I_1, F, G, IE + IE \text{ of } I_1, E \setminus I_1) \quad s, IB', \gamma_2 \vdash spec_2 \Rightarrow I_2}{s, IB, \gamma_1 \cdot \gamma_2 \vdash spec_1 \langle ; \rangle spec_2 \Rightarrow I_1 I_2} \quad (335)$$

Comments:

(323)–(327) 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.

(330) All the necessary sharing has already been checked in the static analysis, therefore this rule needs no premise.

(331),(335) 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 interfaces of global structures as well as of specified structures.

Structure Descriptions

$$\boxed{s, IB, \gamma \vdash strdesc \Rightarrow IE}$$

$$\frac{s, IB, \gamma \vdash sigexp \Rightarrow IS \quad \langle s, IB, \gamma' \vdash strdesc \Rightarrow IE \rangle}{s, IB, \gamma \langle \cdot \gamma' \rangle \vdash strid : sigexp \langle \text{and } strdesc \rangle \Rightarrow \{strid \mapsto IS\} \langle + IE \rangle} \quad (336)$$

Functor Declarations

$$\boxed{s, B, \gamma \vdash fundec \Rightarrow F}$$

$$\frac{s, B, \gamma \vdash funbind \Rightarrow F}{s, B, \gamma \vdash \text{functor } funbind \Rightarrow F} \quad (337)$$

$$\frac{}{s, B, \gamma \vdash \quad \Rightarrow \{ \}} \quad (338)$$

$$\frac{s, B, \gamma_1 \vdash fundec_1 \Rightarrow F_1 \quad s, B + F_1, \gamma_2 \vdash fundec_2 \Rightarrow F_2}{s, B, \gamma_1 \cdot \gamma_2 \vdash fundec_1 \langle ; \rangle fundec_2 \Rightarrow F_1 + F_2} \quad (339)$$

Functor Bindings

$$\boxed{s, B, \gamma \vdash \text{funbind} \Rightarrow F}$$

$$\frac{\begin{array}{c} IB = \text{Inter } B \quad \gamma = \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \langle \cdot \gamma_4 \rangle \\ s, IB, \gamma_1 \vdash \text{psigexp} \Rightarrow \Sigma \quad \Sigma = (N)IS \quad N \cap N \text{ of } IB = \emptyset \\ l = l \text{ of } s \quad s, IB \oplus \{\text{strid} \mapsto IS\}, \gamma_2 \vdash \text{psigexp}' \Rightarrow \Sigma' \\ \langle s, B, \gamma_4 \vdash \text{funbind} \Rightarrow F \rangle \end{array}}{\frac{s \perp, B, \Sigma \vdash \text{strex}^\bullet \Rightarrow \mathcal{R} \quad (s', (\varphi, S)) \in \mathcal{R}}{\exists \mathcal{R}'. s'_l, B \oplus \{\text{strid} \mapsto S\}, \Sigma' \vdash \text{strex} \Rightarrow \mathcal{R}'}} \quad (340)}{s, B, \gamma \vdash \text{funid} (\text{strid} : \text{psigexp}) : \text{psigexp}' = \text{strex} \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (\text{strid}, \text{strex}, B, (N)(IS, \Sigma')) \} \langle + F \rangle}$$

$$\frac{\begin{array}{c} IB = \text{Inter } B \quad \gamma = \gamma_1 \cdot \gamma_2 \langle \cdot \gamma_3 \rangle \\ s, IB, \gamma_1 \vdash \text{psigexp} \Rightarrow \Sigma \quad \Sigma = (N)IS \quad N \cap N \text{ of } IB = \emptyset \\ s, IB \oplus \{\text{strid} \mapsto IS\}, \gamma_2 \vdash \text{psigexp}' \Rightarrow \Sigma' \quad \langle s, B, \gamma_3 \vdash \text{funbind} \Rightarrow F \rangle \end{array}}{s, B, \gamma \vdash \text{funid} (\text{strid} : \text{psigexp}) : \text{psigexp}' = ? \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (\text{strid}, ?, B, (N)(IS, \Sigma')) \} \langle + F \rangle} \quad (341)$$

Comments:

(340) 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.

(340),(341) There is no difference in the result of these two rules: rule 340 simply has an additional check that *strex* satisfies the signature *psigexp'* for any valid functor argument.

Top-level Declarations

$$\boxed{s, B, \gamma \vdash \text{topdec} \Rightarrow \mathcal{B}}$$

$$\frac{s, B, \gamma \vdash \text{strdec} \Rightarrow \mathcal{E}}{s, B, \gamma \vdash \text{strdec} \Rightarrow \{(s', (\text{names } E, E) \text{ in Basis}) \mid (s', E) \in \mathcal{E}\}} \quad (342)$$

$$\frac{s, \text{Inter } B, \gamma \vdash \text{sigdec} \Rightarrow G}{s, B, \gamma \vdash \text{sigdec} \Rightarrow \{(s, (\text{names } G, G) \text{ in Basis})\}} \quad (343)$$

$$\frac{s, B, \gamma \vdash \text{fundec} \Rightarrow F}{s, B, \gamma \vdash \text{fundec} \Rightarrow \{(s, (\text{names } F, F) \text{ in Basis})\}} \quad (344)$$

10 Programs

The phrase classes FullProgram and Program of programs are defined as follows:

$$\begin{aligned} \text{program} & ::= \text{topdec} ; \langle \text{program} \rangle \\ \text{fullprogram} & ::= \text{program} \end{aligned}$$

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 485–488 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 $\vdash \text{fullprogram} \Rightarrow \mathcal{B}, C_{\text{DER}}$. The following main situations can arise:

- There is no $\mathcal{B}, C_{\text{DER}}$ such that $\vdash \text{fullprogram} \Rightarrow \mathcal{B}, C_{\text{DER}}$ 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 $\vdash \text{fullprogram} \Rightarrow \mathcal{B}, C_{\text{DER}}$ 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

$$\boxed{s, B, C_{\text{DER}} \vdash \text{topdec} \Rightarrow \mathcal{B}, C'_{\text{DER}}}$$

$$\frac{\neg \exists C'_{\text{DER}}, \text{topdec}'. C_{\text{DER}} \vdash_{\text{DER}} \text{topdec} \Rightarrow \text{topdec}', C'_{\text{DER}}}{s, B, C_{\text{DER}} \vdash \text{topdec} \Rightarrow \{(s, B)\}, C_{\text{DER}}} \quad (345)$$

$$\frac{C_{\text{DER}} \vdash_{\text{DER}} \text{topdec} \Rightarrow \text{topdec}', C'_{\text{DER}}}{\neg \exists B_{\text{STAT}}, \gamma. \text{Stat } B \vdash_{\text{STAT}} \text{topdec}' \Rightarrow B_{\text{STAT}}, \gamma} \quad (346)$$

$$s, B, C_{\text{DER}} \vdash \text{topdec} \Rightarrow \{(s, B)\}, C_{\text{DER}}$$

$$\frac{
\begin{array}{c}
C_{\text{DER}} \vdash_{\text{DER}} \text{topdec} \Rightarrow \text{topdec}', C'_{\text{DER}} \\
\frac{\text{Stat } B \vdash_{\text{STAT}} \text{topdec}' \Rightarrow B'_{\text{STAT}, \gamma'}}{(\text{names } \gamma \setminus N \text{ of } B) \gamma \succ \gamma'}
\end{array}
\quad
\text{Stat } B \vdash_{\text{STAT}} \text{topdec}' \Rightarrow B_{\text{STAT}, \gamma}
\quad
s, B, \gamma \vdash_{\text{VER}} \text{topdec}' \Rightarrow \mathcal{B}
}{
s, B, C_{\text{DER}} \vdash \text{topdec} \Rightarrow \{(s', B \oplus B') \mid (s', B') \in \mathcal{B}\}, C_{\text{DER}} + C'_{\text{DER}}
}
\quad (347)$$

Comments:

(346) 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.

(347) The rule in the premise ensures that the top-level declaration is verified with a principal trace.

Programs

$$\frac{
\boxed{s, B, C_{\text{DER}} \vdash \text{program} \Rightarrow \mathcal{B}, C'_{\text{DER}}}
\quad
s, B, C_{\text{DER}} \vdash \text{topdec} \Rightarrow \mathcal{B}, C'_{\text{DER}}
\quad
\left\langle \frac{(s', B') \in \mathcal{B}}{\exists \mathcal{B}'. s', B', C'_{\text{DER}} \vdash \text{program} \Rightarrow \mathcal{B}', C''_{\text{DER}}} \right\rangle
}{
s, B, C_{\text{DER}} \vdash \text{topdec} ; \langle \text{program} \rangle \Rightarrow \{(s' \langle ' \rangle, B' \langle ' \rangle) \mid (s', B') \in \mathcal{B}, \langle s', B', C'_{\text{DER}} \vdash \text{program} \Rightarrow \mathcal{B}', C''_{\text{DER}}, (s'', B'') \in \mathcal{B}' \rangle\}, C' \langle ' \rangle_{\text{DER}}
}
\quad (348)$$

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

$$\frac{
\boxed{\vdash \text{fullprogram} \Rightarrow \mathcal{B}, C_{\text{DER}}}
\quad
\frac{s_0, B_0, C_{\text{DER}}^0 \vdash \text{program} \Rightarrow \mathcal{B}, C_{\text{DER}}}{\vdash \text{program} \Rightarrow \mathcal{B}, C_{\text{DER}}}
}{
}
\quad (349)$$

Comment: This rule connects the program semantics with the initial state s_0 , the initial verification basis B_0 , and the initial context for derived forms C_{DER}^0 .

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 ... then ... else ...`”, 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 $id \in \text{Id}$ 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 `< >` enclose optional phrases.
- For any syntax class `X` (over which x ranges) we define the syntax class `Xseq` (over which $xseq$ ranges) as follows:

$$\begin{aligned} xseq ::= x & \quad (\text{singleton sequence}) \\ & \quad (\text{empty sequence}) \\ & (x_1, \dots, x_n) \quad (\text{sequence, } n \geq 1) \end{aligned}$$

Note that the “`...`” 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 exp1 then exp2 else exp3 handle match`

is determined by the above principle. Because `if`-expressions have lower precedence than `handle`-expressions, the sequence parses as

`if exp1 then exp2 else (exp3 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

`if ... then case ... of ... else case ... of ...`

is quite admissible, and will be parsed as

`if ... then (case ... of ...) else (case ... of ...)`

- 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.
- The disambiguation principles (precedence and longest match) do not apply to phrase classes `InfExp` and `InfPat`.
- For any syntax class `X` (over which x ranges) we define the syntax class `X•` (over which x^\bullet ranges) as the same as `X`, except that phrases of class `X•` may not contain `?`.

Disregarding the disambiguation principles for `InfExp` and `InfPat` makes parsing of these classes ambiguous. The derived form semantics (appendix B) ensures that only one of the obtained syntax trees can be translated into a program of the bare language.

<i>atexp</i>	::=	<i>scon</i>	special constant
		$\langle \text{op} \rangle \text{longid}$	long identifier
		$\{ \langle \text{exprow} \rangle \}$	record
		$\# \text{lab}$	record selector
		$()$	0-tuple
		$(\text{exp}_1 , \dots , \text{exp}_n)$	n -tuple, $n \geq 2$
		$[\text{exp}_1 , \dots , \text{exp}_n]$	list, $n \geq 0$
		$(\text{exp}_1 ; \dots ; \text{exp}_n)$	sequence, $n \geq 2$
		let <i>dec</i> in <i>exp</i> ₁ ; ... ; <i>exp</i> _{<i>n</i>} end	local declaration, $n \geq 1$
		(exp)	
		$?$	undefined value
<i>exprow</i>	::=	<i>lab</i> = <i>exp</i> $\langle , \text{exprow} \rangle$	expression row
<i>appexp</i>	::=	<i>atexp</i>	
		<i>appexp</i> <i>atexp</i>	application expression
<i>infexp</i>	::=	<i>appexp</i>	
		<i>infexp</i> ₁ <i>id</i> <i>infexp</i> ₂	infix expression
<i>exp</i>	::=	<i>infexp</i>	
		<i>exp</i> : <i>ty</i>	typed
		$\text{exp}_1^\bullet == \text{exp}_2^\bullet$	comparison (R)
		$\text{exp}_1^\bullet \neq \text{exp}_2^\bullet$	comparison (R)
		exists <i>match</i> [•]	existential quantifier
		forall <i>match</i> [•]	universal quantifier
		<i>exp</i> [•] terminates	convergence predicate
		<i>exp</i> [•] proper	definedness predicate
		<i>exp</i> ₁ andalso <i>exp</i> ₂	conjunction
		<i>exp</i> ₁ orelse <i>exp</i> ₂	disjunction
		<i>exp</i> ₁ implies <i>exp</i> ₂	implication (R)
		<i>exp</i> ₁ iff <i>exp</i> ₂	if and only if
		<i>exp</i> handle <i>match</i>	handle exception
		raise <i>exp</i>	raise exception
		<i>exp</i> raises <i>match</i>	test for exception
		<i>exp</i> raises <i>pat</i>	test for exception
		if <i>exp</i> ₁ then <i>exp</i> ₂ else <i>exp</i> ₃	conditional
		case <i>exp</i> of <i>match</i>	case analysis
		fn <i>match</i>	function
<i>match</i>	::=	<i>mrule</i> $\langle \text{match} \rangle$	
<i>mrule</i>	::=	<i>pat</i> => <i>exp</i>	

Figure 19: Grammar: Expressions and Matches

<i>dec</i>	<pre> ::= val <i>valbind</i> fun <i>fvalbind</i> type <i>typbind</i> eqtype <i>typbind</i> datatype <i>datbind</i> <withtype <i>typbind</i>[•]> abstype <i>datbind</i> <withtype <i>typbind</i>[•]> with <i>dec</i> end exception <i>exbind</i> local <i>dec</i>₁ in <i>dec</i>₂ end open <i>longid</i>₁...<i>longid</i>_{<i>n</i>} <i>dec</i>₁ <;> <i>dec</i>₂ infix <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} infixr <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} nonfix <i>id</i>₁ ... <i>id</i>_{<i>n</i>} </pre>	<p>value declaration function declaration type declaration equality type declaration datatype declaration abstype declaration exception declaration local declaration open declaration, $n \geq 1$ sequential declaration empty declaration infix directive, $n \geq 1$ infix directive, $n \geq 1$ nonfix directive, $n \geq 1$</p>
<i>valbind</i>	<pre> ::= <i>pat</i> = <i>exp</i> <and <i>valbind</i>> rec <i>valbind</i> </pre>	
<i>fvalbind</i>	<pre> ::= <i>funcbind</i> <and <i>fvalbind</i>> </pre>	function declarations
<i>funcbind</i>	<pre> ::= <i>fpat</i> <: <i>ty</i>> = <i>exp</i> < <i>funcbind</i>> </pre>	single function
<i>typbind</i>	<pre> ::= <i>tyvarseq id</i> = <i>ty</i> <and <i>typbind</i>> <i>tyvarseq id</i> = ? <and <i>typbind</i>> </pre>	<p>type binding question mark type binding</p>
<i>datbind</i>	<pre> ::= <i>tyvarseq id</i> = <i>conbind</i> <and <i>datbind</i>> </pre>	
<i>conbind</i>	<pre> ::= <i>id</i> <of <i>ty</i>> < <i>conbind</i>> </pre>	
<i>exbind</i>	<pre> ::= <i>id</i> <of <i>ty</i>> <and <i>exbind</i>> <i>id</i> = <i>longid</i> <and <i>exbind</i>> </pre>	

Figure 20: Grammar: Declarations and Bindings

$atpat$	$::=$	$-$ $scon$ $\langle op \rangle longid$ $\{ \langle patrow \rangle \}$ $()$ (pat_1, \dots, pat_n) $[pat_1, \dots, pat_n]$ (pat)	wildcard special constant record 0-tuple n -tuple, $n \geq 2$ list, $n \geq 0$
$patrow$	$::=$	\dots $lab = pat \langle , patrow \rangle$ $id \langle : ty \rangle \langle as pat \rangle \langle , patrow \rangle$	wildcard pattern row label as variable
$apppat$	$::=$	$\langle op \rangle longid atpat$ $atpat$	construction atomic
$infpat$	$::=$	$apppat$ $infpat_1 id infpat_2$	application or atomic infix construction
pat	$::=$	$infpat$ $pat : ty$ $\langle op \rangle id \langle : ty \rangle as pat$	typed layered
$fpat$	$::=$	$\langle op \rangle id atpat_1 \dots atpat_n$ $(atpat_1 id atpat'_1) atpat_2 \dots atpat_n$ $atpat_1 id atpat'_1$	$fvalbind$ pattern sequence

Figure 21: Grammar: Patterns

ty	$::=$	$tyvar$ $\{ \langle tyrow \rangle \}$ $tyseq longid$ $ty_1 * \dots * ty_n$ $ty \rightarrow ty'$ (ty)	type variable record type type construction tuple type, $n \geq 2$ function type (R)
$tyrow$	$::=$	$lab : ty \langle , tyrow \rangle$	type-expression row

Figure 22: Grammar: Type expressions

<i>strexp</i>	::= struct <i>strdec</i> end <i>longid</i> <i>id</i> (<i>strexp</i>) <i>id</i> (<i>strdec</i>) let <i>strdec</i> in <i>strexp</i> end	generative structure identifier functor application local declaration
<i>strdec</i>	::= <i>dec</i> axiom <i>ax</i> structure <i>strbind</i> local <i>strdec</i> ₁ in <i>strdec</i> ₂ end <i>strdec</i> ₁ <;> <i>strdec</i> ₂	declaration axiom structure local sequential empty
<i>ax</i>	::= <i>axexp</i> < and <i>ax</i> >	axioms
<i>axexp</i>	::= <i>exp</i> [•]	axiomatic expression
<i>strbind</i>	::= <i>sglstrbind</i> < and <i>strbind</i> >	structure binding
<i>sglstrbind</i>	::= <i>id</i> : <i>psigexp</i> = <i>strexp</i> <i>id</i> : <i>psigexp</i> = ? <i>id</i> = <i>strexp</i>	single structure binding undefined structure binding unguarded structure binding
<i>sigexp</i>	::= sig <i>spec</i> end <i>id</i>	generative signature identifier
<i>psigexp</i>	::= <i>sigexp</i>	principal signature
<i>sigdec</i>	::= signature <i>sigbind</i> <i>sigdec</i> ₁ <;> <i>sigdec</i> ₂	single sequential empty
<i>sigbind</i>	::= <i>id</i> = <i>psigexp</i> < and <i>sigbind</i> >	

Figure 23: Grammar: Structure and Signature Expressions

<i>spec</i>	<code>::= val <i>valdesc</i> type <i>typdesc</i> eqtype <i>typdesc</i> datatype <i>datdesc</i> exception <i>exdesc</i> axiom <i>axdesc</i> structure <i>strdesc</i> sharing <i>shareq</i> local <i>spec</i>₁ in <i>spec</i>₂ end open <i>longid</i>₁...<i>longid</i>_{<i>n</i>} include <i>id</i>₁...<i>id</i>_{<i>n</i>} <i>spec</i>₁ <;> <i>spec</i>₂</code>	value type eqtype datatype exception axiom structure sharing local open ($n \geq 1$) include ($n \geq 1$) sequential empty
<i>valdesc</i>	<code>::= <i>id</i> : <i>ty</i> <and <i>valdesc</i>></code>	
<i>typdesc</i>	<code>::= <i>tyvarseq id</i> <and <i>typdesc</i>></code>	
<i>datdesc</i>	<code>::= <i>tyvarseq id = condesc</i> <and <i>datdesc</i>></code>	
<i>condesc</i>	<code>::= <i>id</i> <of <i>ty</i>> < <i>condesc</i>></code>	
<i>exdesc</i>	<code>::= <i>id</i> <of <i>ty</i>> <and <i>exdesc</i>></code>	
<i>axdesc</i>	<code>::= <i>specexp</i> <and <i>axdesc</i>></code>	
<i>specexp</i>	<code>::= let <i>strdec</i> in <i>axexp</i> end <i>exp</i>[•]</code>	
<i>strdesc</i>	<code>::= <i>id</i> : <i>sigexp</i> <and <i>strdesc</i>></code>	
<i>shareq</i>	<code>::= <i>longid</i>₁ = ... = <i>longid</i>_{<i>n</i>} <code>type <i>longid</i>₁ = ... = <i>longid</i>_{<i>n</i>} <i>shareq</i>₁ and <i>shareq</i>₂</code></code>	structure sharing ($n \geq 2$) type sharing ($n \geq 2$) multiple

Figure 24: Grammar: Specifications

<i>fundec</i>	$::=$ functor <i>funbind</i> <i>fundec</i> ₁ $\langle ; \rangle$ <i>fundec</i> ₂	single sequence empty
<i>funbind</i>	$::=$ <i>id</i> (<i>id</i> ' : <i>psigexp</i>) : <i>psigexp</i> ' = <i>strexp</i> \langle and <i>funbind</i> \rangle <i>id</i> (<i>spec</i>) : <i>psigexp</i> = <i>strexp</i> \langle and <i>funbind</i> \rangle <i>id</i> (<i>id</i> ' : <i>psigexp</i>) : <i>psigexp</i> ' = ? \langle and <i>funbind</i> \rangle <i>id</i> (<i>spec</i>) : <i>psigexp</i> = ? \langle and <i>funbind</i> \rangle	functor binding undefined functor binding
<i>topdec</i>	$::=$ <i>strdec</i> <i>sigdec</i> <i>fundec</i> <i>exp</i>	structure-level declaration signature declaration functor declaration expression at top-level
<i>program</i>	$::=$ <i>topdec</i> ; \langle <i>program</i> \rangle	
<i>fullprogram</i>	$::=$ <i>program</i>	

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.

$$\begin{aligned}
st &\in \text{Status} = \{\mathbf{v}, \mathbf{c}, \mathbf{e}, \mathbf{g}, \mathbf{s}, \mathbf{f}, \mathbf{t}\} \\
fix &\in \text{Fixity} = \{\mathbf{l}, \mathbf{r}, \mathbf{n}\} \\
n &\in \mathcal{N} = \{0, 1, 2, \dots\} \\
VE &\in \text{VarEnv} = \text{Id} \xrightarrow{\text{fin}} (\mathcal{N} \times \text{Fixity}) \\
StE &\in \text{StatEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Status} \\
TE &\in \text{TyEnv} = \text{Id} \xrightarrow{\text{fin}} (\text{TypeFcn} \uplus \{\text{FAIL}\}) \\
\Lambda\alpha^{(k)}.ty &\in \text{TypeFcn} = \uplus_{k \geq 0} \text{TyVar}^k \times \text{Ty} \\
SE &\in \text{StrEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Env} \\
F &\in \text{FunEnv} = \text{Id} \xrightarrow{\text{fin}} (\text{Env} \times \text{Env}) \\
G &\in \text{SigEnv} = \text{Id} \xrightarrow{\text{fin}} \text{Env} \\
(SE, TE, StE) \text{ or } E &\in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{StatEnv} \\
(E, VE, F, G) \text{ or } C &\in \text{Context} = \text{Env} \times \text{VarEnv} \times \text{FunEnv} \times \text{SigEnv} \\
idval &\in \text{Idval} = \text{Env} \uplus \text{TypeFcn} \uplus \text{Status} \uplus \\
&\quad (\mathcal{N} \times \text{Fixity}) \uplus (\text{Env} \times \text{Env}) \uplus \{\text{FAIL}\} \\
\pi &\in \text{Context} \rightarrow (\text{Id} \xrightarrow{\text{fin}} \text{Idval})
\end{aligned}$$

Figure 26: Semantic Objects for Derived Forms

The results of looking up a long identifier in a context (rules 350–352) 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 351. 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. $(SE \text{ of } E \text{ of } _)$ selects the structure environment of a context.

Given a type expression ty , a sequence of type variables $\alpha_1, \dots, \alpha_k$ and a sequence of type expressions ty_1, \dots, ty_k , we write $ty\{ty_1/\alpha_1, \dots, ty_k/\alpha_k\}$ 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:

$$\begin{aligned} [StE](id) &= StE(id) && \text{if } id \in \text{Dom } StE \\ &= \mathbf{v} && \text{otherwise} \\ [VE](id) &= VE(id) && \text{if } id \in \text{Dom } VE \\ &= (0, \mathbf{n}) && \text{otherwise} \end{aligned}$$

In other words: $[StE]$ resp. $[VE]$ is an (infinite) map which extends the finite map StE resp. VE by the default value \mathbf{v} resp. $(0, \mathbf{n})$.

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 \mathbf{v} , \mathbf{c} , \mathbf{e} , \mathbf{t} , \mathbf{s} , \mathbf{g} and \mathbf{f} refer to `Var`, `Con`, `ExCon`, `TyCon`, `StrId`, `SigId` and `FunId`, respectively. Thus, any value variable var has the form $id^{\mathbf{v}}$ with $id \in \text{Id}$, 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 (\mathbf{l} or \mathbf{r} , respectively) or non-infix (\mathbf{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 \bar{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 $st \succ st' \iff st = st' \vee st' = \mathbf{v}$. We extend this to (status, structure) environments as follows:

$$\begin{aligned} E_1 \succ E_2 &\iff StE \text{ of } E_1 \succ StE \text{ of } E_2 \wedge SE \text{ of } E_1 \succ SE \text{ of } E_2 \\ SE_1 \succ SE_2 &\iff \forall id \in \text{Dom } SE_2. id \in \text{Dom } SE_1 \wedge SE_1(id) \succ SE_2(id) \\ StE_1 \succ StE_2 &\iff \forall id \in \text{Dom } StE_2. id \in \text{Dom } StE_1 \wedge StE_1(id) \succ StE_2(id) \end{aligned}$$

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 $C, A_1^* \vdash \textit{phrase} \Rightarrow \textit{phrase}', A_2^*$, then *phrase'* and A_2^* are determined by C, A_1^* , and *phrase*, modulo choice of fresh identifier names (e.g. as in rule 357).

We do not give all rules explicitly. When we omit the sentence form for a phrase class *Phrase*, the sentence form is $C \vdash \textit{phrase} \Rightarrow \textit{phrase}'$. 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

$$\boxed{C, \pi, st \vdash \textit{longid} \Rightarrow \textit{longid}', \textit{idval}}$$

$$\frac{id \in \text{Dom}(\pi C) \quad \textit{idval} = \pi(C)(id)}{C, \pi, st \vdash id \Rightarrow id^{st}, \textit{idval}} \quad (350)$$

$$\frac{id \notin \text{Dom}(\pi C)}{C, \pi, st \vdash id \Rightarrow id^{st}, \text{FAIL}} \quad (351)$$

$$\frac{(SE \text{ of } E \text{ of } C)(id) = E \quad C \oplus E, \pi, st \vdash \textit{longid} \Rightarrow \textit{longid}', \textit{idval}}{C, \pi, st \vdash id.\textit{longid} \Rightarrow id^{\mathbf{S}}.\textit{longid}', \textit{idval}} \quad (352)$$

Comments:

(351) 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 357.

Value Identifiers

$$\boxed{C \vdash \langle \text{op} \rangle \textit{longid} \Rightarrow \textit{longid}', st}$$

$$\frac{[VE \text{ of } C](id) = (n, \textit{fix}) \quad \textit{fix} \neq \mathbf{n} \quad [StE \text{ of } E \text{ of } C](id) = st}{C \vdash \text{op } id \Rightarrow id^{st}, st} \quad (353)$$

$$\frac{[VE \text{ of } C](id) = (n, \mathbf{n}) \quad [StE \text{ of } E \text{ of } C](id) = st}{C \vdash \langle \text{op} \rangle id \Rightarrow id^{st}, st} \quad (354)$$

$$\frac{(SE \text{ of } E \text{ of } C)(id) = E \quad C \oplus E \vdash \text{op } \textit{longid} \Rightarrow \textit{longid}', st}{C \vdash \langle \text{op} \rangle id.\textit{longid} \Rightarrow id^{\mathbf{S}}.\textit{longid}', st} \quad (355)$$

Comments:

- (355) 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

$$\boxed{C \vdash atexp \Rightarrow atexp'}$$

$$\frac{C \vdash \langle \text{op} \rangle longid \Rightarrow longid', st}{C \vdash \langle \text{op} \rangle longid \Rightarrow longid'} \quad (356)$$

$$\frac{C, StE \text{ of } E \text{ of } -, v \vdash id \Rightarrow var, FAIL}{C \vdash \# lab \Rightarrow (\text{fn } \{ lab=var, \dots \} \Rightarrow var)} \quad (357)$$

$$\overline{C \vdash () \Rightarrow \{ \}} \quad (358)$$

$$\frac{n \geq 2 \quad C \vdash exp_1 \Rightarrow exp'_1 \quad \dots \quad C \vdash exp_n \Rightarrow exp'_n}{C \vdash (exp_1, \dots, exp_n) \Rightarrow \{ 1=exp'_1, \dots, \bar{n}=exp'_n \}} \quad (359)$$

$$\frac{n \geq 0 \quad C \vdash (exp_1) :: \dots :: (exp_n) :: nil \Rightarrow exp}{C \vdash [exp_1, \dots, exp_n] \Rightarrow (exp)} \quad (360)$$

$$\frac{n \geq 1 \quad C \vdash \text{case } (exp_1) \text{ of } _ \Rightarrow \dots \text{case } (exp_n) \text{ of } _ \Rightarrow (exp) \Rightarrow exp'}{C \vdash (exp_1; \dots; exp_n; exp) \Rightarrow (exp')} \quad (361)$$

$$\frac{n \geq 1 \quad C \vdash dec \Rightarrow dec', C' \quad C + C' \vdash (exp_1; \dots; exp_n) \Rightarrow atexp}{C \vdash \text{let } dec \text{ in } exp_1; \dots; exp_n \text{ end} \Rightarrow \text{let } dec' \text{ in } atexp \text{ end}} \quad (362)$$

Comments:

- (357) The premise selects a fresh identifier *id* and attributes it with status *v*, giving the value variable *var*.
- (360) It is assumed here that neither **nil** nor **::** has been rebound, and **::** still has right-associative infix status.
- (362) The side-condition $n \geq 1$ means that the rule applies to all **let**-expressions.

Applications

$$\boxed{C \vdash appexp \Rightarrow exp}$$

$$\frac{C \vdash appexp \Rightarrow exp \quad C \vdash atexp \Rightarrow atexp'}{C \vdash appexp atexp \Rightarrow (exp) atexp'} \quad (363)$$

Infix Expressions

$$\boxed{C, n, fix \vdash infexp \Rightarrow exp}$$

$$\frac{C \vdash appexp \Rightarrow exp}{C, n, fix \vdash appexp \Rightarrow exp} \quad (364)$$

$$\frac{\begin{array}{l} [VE \text{ of } C](id) = (n', 1) \quad n < n' \vee (n = n' \wedge fix = 1) \\ [StE \text{ of } E \text{ of } C](id) = st \\ C, n', 1 \vdash infexp_1 \Rightarrow exp_1 \quad C, n', n \vdash infexp_2 \Rightarrow exp_2 \end{array}}{C, n, fix \vdash infexp_1 \ id \ infexp_2 \Rightarrow id^{st} \{1=exp_1, 2=exp_2\}} \quad (365)$$

$$\frac{\begin{array}{l} [VE \text{ of } C](id) = (n', r) \quad n < n' \vee (n = n' \wedge fix = r) \\ [StE \text{ of } E \text{ of } C](id) = st \\ C, n', n \vdash infexp_1 \Rightarrow exp_1 \quad C, n', r \vdash infexp_2 \Rightarrow exp_2 \end{array}}{C, n, fix \vdash infexp_1 \ id \ infexp_2 \Rightarrow id^{st} \{1=exp_1, 2=exp_2\}} \quad (366)$$

Comments:

(366) 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

$$\boxed{C \vdash exp \Rightarrow exp'}$$

Not all of the parentheses these rules generate are really necessary; many of them are here only for uniformity of presentation.

$$\frac{C, 0, fix \vdash infexp \Rightarrow exp}{C \vdash infexp \Rightarrow (exp)} \quad (367)$$

$$\frac{C \vdash \text{not}(exp_1^\bullet == exp_2^\bullet) \Rightarrow exp}{C \vdash exp_1^\bullet \neq exp_2^\bullet \Rightarrow (exp)} \quad (368)$$

$$\frac{C \vdash (exp^\bullet \text{ terminates}) \text{ andalso not}(exp^\bullet \text{ raises } _) \Rightarrow exp}{C \vdash exp^\bullet \text{ proper} \Rightarrow (exp)} \quad (369)$$

$$\frac{C \vdash \text{if } exp_1 \text{ then } exp_2 \text{ else false} \Rightarrow exp}{C \vdash exp_1 \text{ andalso } exp_2 \Rightarrow (exp)} \quad (370)$$

$$\frac{C \vdash \text{if } exp_1 \text{ then true else } exp_2 \Rightarrow exp}{C \vdash exp_1 \text{ orelse } exp_2 \Rightarrow (exp)} \quad (371)$$

$$\frac{C \vdash \text{if } exp_1 \text{ then } exp_2 \text{ else true} \Rightarrow exp}{C \vdash exp_1 \text{ implies } exp_2 \Rightarrow (exp)} \quad (372)$$

$$\frac{C \vdash \text{if } exp_1 \text{ then } exp_2 \text{ else not}(exp_2) \Rightarrow exp}{C \vdash exp_1 \text{ iff } exp_2 \Rightarrow (exp)} \quad (373)$$

$$\frac{C \vdash ((exp ; false) \text{ handle } match) \text{ handle } _ \Rightarrow false \Rightarrow exp'}{C \vdash exp \text{ raises } match \Rightarrow (exp')} \quad (374)$$

$$\frac{C \vdash exp \text{ raises } pat \Rightarrow true \Rightarrow exp'}{C \vdash exp \text{ raises } pat \Rightarrow (exp')} \quad (375)$$

$$\frac{C \vdash \text{ case } exp_1 \text{ of } true \Rightarrow (exp_2) \mid false \Rightarrow (exp_3) \Rightarrow exp}{C \vdash \text{ if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \Rightarrow (exp)} \quad (376)$$

$$\frac{C \vdash (\text{fn } match)(exp) \Rightarrow exp'}{C \vdash \text{ case } exp \text{ of } match \Rightarrow (exp')} \quad (377)$$

Comments:

(367) 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.

(368),(369),(373) It is assumed here that **not** has not been rebound, neither in *StE* nor in *VE*.

(370)–(376) It is assumed here that **true** and **false** have not been rebound.

Match Rules

$$\boxed{C \vdash mrule \Rightarrow mrule'}$$

$$\frac{C \vdash pat \Rightarrow pat', StE \quad C + StE \vdash exp \Rightarrow exp'}{C \vdash pat \Rightarrow exp \Rightarrow pat' \Rightarrow exp'} \quad (378)$$

Declarations

$$\boxed{C \vdash dec \Rightarrow dec', C'}$$

$$\frac{C \vdash valbind \Rightarrow valbind', StE}{C \vdash \text{val } valbind \Rightarrow \text{val } valbind', StE \text{ in Env in Context}} \quad (379)$$

$$\frac{C + StE \vdash fvalbind \Rightarrow valbind, StE}{C \vdash \text{fun } fvalbind \Rightarrow \text{val rec } valbind, StE \text{ in Env in Context}} \quad (380)$$

$$\frac{C \vdash typbind \Rightarrow typbind', TE, TE'}{C \vdash \text{type } typbind \Rightarrow \text{type } typbind', TE' \text{ in Env in Context}} \quad (381)$$

$$\frac{C \vdash typbind \Rightarrow typbind', TE, TE'}{C \vdash \text{eqtype } typbind \Rightarrow \text{eqtype } typbind', TE' \text{ in Env in Context}} \quad (382)$$

$$\begin{array}{c}
\langle C + E \vdash \text{typbind}^\bullet \Rightarrow \text{typbind}', TE, TE' \rangle \\
C + E \langle +TE \rangle \vdash \text{datbind} \Rightarrow \text{datbind}', E \\
\hline
C \vdash \text{datatype } \text{datbind} \langle \text{withtype } \text{typbind}^\bullet \rangle \Rightarrow \\
\text{datatype } \text{datbind}' \langle ; \text{type } \text{typbind}' \rangle, E \langle +TE' \rangle \text{ in Context}
\end{array} \quad (383)$$

$$\begin{array}{c}
\langle C + E \vdash \text{typbind}^\bullet \Rightarrow \text{typbind}', TE, TE' \rangle \\
C + E \langle +TE \rangle \vdash \text{datbind} \Rightarrow \text{datbind}', E \\
C + E \langle +TE' \rangle \vdash \text{dec} \Rightarrow \text{dec}', C' \\
\hline
C \vdash \text{abstype } \text{datbind} \langle \text{withtype } \text{typbind}^\bullet \rangle \text{ with } \text{dec} \text{ end} \Rightarrow \\
\text{abstype } \text{datbind}' \text{ with } \langle \text{type } \text{typbind}' ; \rangle \text{dec}' \text{ end}, \\
\langle TE' \text{ in Context } + \rangle C'
\end{array} \quad (384)$$

$$\begin{array}{c}
C \vdash \text{exbind} \Rightarrow \text{exbind}', StE \\
\hline
C \vdash \text{exception } \text{exbind} \Rightarrow \text{exception } \text{exbind}', StE \text{ in Env in Context}
\end{array} \quad (385)$$

$$\begin{array}{c}
C \vdash \text{dec}_1 \Rightarrow \text{dec}'_1, C_1 \quad C + C_1 \vdash \text{dec}_2 \Rightarrow \text{dec}'_2, C_2 \\
\hline
C \vdash \text{local } \text{dec}_1 \text{ in } \text{dec}_2 \text{ end} \Rightarrow \text{local } \text{dec}'_1 \text{ in } \text{dec}'_2 \text{ end}, C_2
\end{array} \quad (386)$$

$$\begin{array}{c}
C, SE \text{ of } E \text{ of } _, \mathbf{s} \vdash \text{longid}_1 \Rightarrow \text{longid}'_1, E_1 \\
\quad \dots \\
C, SE \text{ of } E \text{ of } _, \mathbf{s} \vdash \text{longid}_n \Rightarrow \text{longid}'_n, E_n \\
\hline
C \vdash \text{open } \text{longid}_1 \dots \text{longid}_n \Rightarrow \text{open } \text{longid}'_1 \dots \text{longid}'_n, E_1 + \dots + E_n \text{ in Context}
\end{array} \quad (387)$$

$$\begin{array}{c}
\hline
C \vdash \quad \Rightarrow \quad , \{ \} \text{ in Context}
\end{array} \quad (388)$$

$$\begin{array}{c}
C \vdash \text{dec}_1 \Rightarrow \text{dec}'_1, C_1 \quad C + C_1 \vdash \text{dec}_2 \Rightarrow \text{dec}'_2, C_2 \\
\hline
C \vdash \text{dec}_1 \langle ; \rangle \text{dec}_2 \Rightarrow \text{dec}'_1 \langle ; \rangle \text{dec}'_2, C_1 + C_2
\end{array} \quad (389)$$

$$\begin{array}{c}
n \geq 1 \quad VE = \{ id \mapsto (0 \langle +d \rangle, \mathbf{1}) ; id \in \{ id_1, \dots, id_n \} \} \\
\hline
C \vdash \text{infix } \langle d \rangle id_1 \dots id_n \Rightarrow \quad , VE \text{ in Context}
\end{array} \quad (390)$$

$$\begin{array}{c}
n \geq 1 \quad VE = \{ id \mapsto (0 \langle +d \rangle, \mathbf{r}) ; id \in \{ id_1, \dots, id_n \} \} \\
\hline
C \vdash \text{infixr } \langle d \rangle id_1 \dots id_n \Rightarrow \quad , VE \text{ in Context}
\end{array} \quad (391)$$

$$\begin{array}{c}
n \geq 1 \quad VE = \{ id \mapsto (0, \mathbf{n}) ; id \in \{ id_1, \dots, id_n \} \} \\
\hline
C \vdash \text{nonfix } id_1 \dots id_n \Rightarrow \quad , VE \text{ in Context}
\end{array} \quad (392)$$

Comments:

(383),(384) Type declarations of these forms are mutually recursive.

(390)–(392) Infix directives are replaced by the empty declaration. The digit d is treated in the premise as the corresponding natural number.

Value Bindings

$$\boxed{C \vdash \text{valbind} \Rightarrow \text{valbind}', StE}$$

$$\frac{C \vdash \text{pat} \Rightarrow \text{pat}', StE \quad C \vdash \text{exp} \Rightarrow \text{exp}' \quad \langle C \vdash \text{valbind} \Rightarrow \text{valbind}', StE' \rangle}{C \vdash \text{pat} = \text{exp} \langle \text{and valbind} \rangle \Rightarrow \text{pat}' = \text{exp}' \langle \text{and valbind}' \rangle, StE \langle + StE' \rangle} \quad (393)$$

$$\frac{C + StE \vdash \text{valbind} \Rightarrow \text{valbind}', StE}{C \vdash \text{rec valbind} \Rightarrow \text{rec valbind}', StE} \quad (394)$$

Comments:

(394) 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

$$\boxed{C \vdash \text{fvalbind} \Rightarrow \text{valbind}, StE}$$

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.

$$\frac{\begin{array}{l} C \vdash \text{funcbind} \Rightarrow \text{match}, \text{var}, n \quad \text{var} = \text{id}^{\mathbf{v}} \\ \forall i, j. 1 \leq i, j \leq n \wedge i \neq j \implies \text{id}_i \neq \text{id}_j \\ \forall i. 1 \leq i \leq n \implies C, StE \text{ of } E \text{ of } -, \mathbf{v} \vdash \text{id}_i \Rightarrow \text{var}_i, \text{FAIL} \\ C \vdash \text{case } (\text{id}_1, \dots, \text{id}_n) \text{ of } \text{match} \Rightarrow \text{exp} \\ \langle C \vdash \text{fvalbind} \Rightarrow \text{valbind}, StE \rangle \end{array}}{C \vdash \text{funcbind} \langle \text{and fvalbind} \rangle \Rightarrow \text{var} = \text{fn var}_1 \Rightarrow \dots \text{fn var}_n \Rightarrow \text{exp} \langle \text{and valbind} \rangle, \{\text{id} \mapsto \mathbf{v}\} \langle + StE \rangle} \quad (395)$$

Single Function Bindings

$$\boxed{C \vdash \text{funcbind} \Rightarrow \text{match}, \text{var}, n}$$

$$\frac{\begin{array}{l} C \vdash \text{fpat} \Rightarrow \text{pat}, \text{var}, n, StE \quad \langle C \vdash \text{ty} \Rightarrow \text{ty}' \rangle \\ C + StE \vdash \text{exp} \Rightarrow \text{exp}' \quad \langle \langle C \vdash \text{funcbind} \Rightarrow \text{match}, \text{var}, n \rangle \rangle \end{array}}{C \vdash \text{fpat} \langle : \text{ty} \rangle = \text{exp} \langle \langle | \text{funcbind} \rangle \rangle \Rightarrow \text{pat} \Rightarrow \text{exp}' \langle : \text{ty}' \rangle \langle \langle | \text{match} \rangle \rangle, \text{var}, n} \quad (396)$$

Comment: Notice that the function name and the number of parameters have to be the same throughout a *funcbind*.

Type Bindings

$$\boxed{C \vdash \text{typbind} \Rightarrow \text{typbind}', TE, TE'}$$

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 430.

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C \vdash \text{ty} \Rightarrow \text{ty}' \quad \langle C \vdash \text{typbind} \Rightarrow \text{typbind}', TE, TE' \rangle}{C \vdash \text{tyvarseq } id = \text{ty} \langle \text{and } \text{typbind} \rangle \Rightarrow \text{tyvarseq } id^{\text{t}} = \text{ty}' \langle \text{and } \text{typbind}' \rangle, \{id \mapsto \Lambda \alpha^{(k)}. \text{ty}'\} \langle +TE \rangle, \{id \mapsto \text{FAIL}\} \langle +TE' \rangle} \quad (397)$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad \langle C \vdash \text{typbind} \Rightarrow \text{typbind}', TE, TE' \rangle}{C \vdash \text{tyvarseq } id = ? \langle \text{and } \text{typbind} \rangle \Rightarrow \text{tyvarseq } id^{\text{t}} = ? \langle \text{and } \text{typbind}' \rangle, \{id \mapsto \text{FAIL}\} \langle +TE \rangle, \{id \mapsto \text{FAIL}\} \langle +TE' \rangle} \quad (398)$$

Data Type Bindings

$$\boxed{C \vdash \text{datbind} \Rightarrow \text{datbind}', E}$$

$$\frac{C \vdash \text{conbind} \Rightarrow \text{conbind}', StE \quad \langle C \vdash \text{datbind} \Rightarrow \text{datbind}', E \rangle}{C \vdash \text{tyvarseq } id = \text{conbind} \langle \text{and } \text{datbind} \rangle \Rightarrow \text{tyvarseq } id^{\text{t}} = \text{conbind}' \langle \text{and } \text{datbind}' \rangle, (\{id \mapsto \text{FAIL}\}, StE) \text{ in Env } \langle +E \rangle} \quad (399)$$

Constructor Bindings

$$\boxed{C \vdash \text{conbind} \Rightarrow \text{conbind}', StE}$$

$$\frac{\langle C \vdash \text{conbind} \Rightarrow \text{conbind}', StE \rangle}{C \vdash id \langle | \text{conbind} \rangle \Rightarrow id^{\text{c}} \langle | \text{conbind}' \rangle, \{id \mapsto \text{c}\} \langle +StE \rangle} \quad (400)$$

$$\frac{C \vdash \text{ty} \Rightarrow \text{ty}' \quad \langle C \vdash \text{conbind} \Rightarrow \text{conbind}', StE \rangle}{C \vdash id \text{ of } \text{ty} \langle | \text{conbind} \rangle \Rightarrow id^{\text{c}} \text{ of } \text{ty}' \langle | \text{conbind}' \rangle, \{id \mapsto \text{c}\} \langle +StE \rangle} \quad (401)$$

Exception Bindings

$$\boxed{C \vdash \text{exbind} \Rightarrow \text{exbind}', StE}$$

$$\frac{\langle C \vdash \text{ty} \Rightarrow \text{ty}' \rangle \quad \langle \langle C \vdash \text{exbind} \Rightarrow \text{exbind}', StE \rangle \rangle}{C \vdash id \langle \text{of } \text{ty} \rangle \langle \langle \text{and } \text{exbind} \rangle \rangle \Rightarrow id^{\text{e}} \langle \text{of } \text{ty}' \rangle \langle \langle \text{and } \text{exbind}' \rangle \rangle, \{id \mapsto \text{e}\} \langle \langle +StE \rangle \rangle} \quad (402)$$

$$\frac{C, StE \text{ of } E \text{ of } _, \text{e} \vdash \text{longid} \Rightarrow \text{longexcon}, \text{e} \quad \langle C \vdash \text{exbind} \Rightarrow \text{exbind}', StE \rangle}{C \vdash id = \text{longid} \langle \text{and } \text{exbind} \rangle \Rightarrow id^{\text{e}} = \text{longexcon} \langle \text{and } \text{exbind}' \rangle, \{id \mapsto \text{e}\} \langle +StE \rangle} \quad (403)$$

Atomic Patterns

$$\boxed{C \vdash \text{atpat} \Rightarrow \text{atpat}', StE}$$

$$\overline{C \vdash _ \Rightarrow _, \{ \}} \quad (404)$$

$$\overline{C \vdash \text{scon} \Rightarrow \text{scon}, \{ \}} \quad (405)$$

$$\frac{C \vdash \langle \text{op} \rangle \text{id} \Rightarrow \text{var}, \mathbf{v}}{C \vdash \langle \text{op} \rangle \text{id} \Rightarrow \text{var}, \{ \text{id} \mapsto \mathbf{v} \}} \quad (406)$$

$$\frac{C \vdash \langle \text{op} \rangle \text{longid} \Rightarrow \text{longid}', st \quad st \neq \mathbf{v}}{C \vdash \langle \text{op} \rangle \text{longid} \Rightarrow \text{longid}', \{ \}} \quad (407)$$

$$\frac{\langle C \vdash \text{patrow} \Rightarrow \text{patrow}', StE \rangle}{C \vdash \{ \langle \text{patrow} \rangle \} \Rightarrow \{ \langle \text{patrow}' \rangle \}, \{ \} \langle + StE \rangle} \quad (408)$$

$$\overline{C \vdash () \Rightarrow \{ \}, \{ \}} \quad (409)$$

$$\frac{n \geq 2 \quad C \vdash \text{pat}_1 \Rightarrow \text{pat}'_1, StE_1 \quad \dots \quad C \vdash \text{pat}_n \Rightarrow \text{pat}'_n, StE_n}{C \vdash (\text{pat}_1, \dots, \text{pat}_n) \Rightarrow \{ \mathbf{1} = \text{pat}_1, \dots, \bar{n} = \text{pat}_n \}, StE_1 + \dots + StE_n} \quad (410)$$

$$\frac{n \geq 0 \quad C \vdash (\text{pat}_1) :: \dots :: (\text{pat}_n) :: \text{nil} \Rightarrow \text{pat}, StE}{C \vdash [\text{pat}_1, \dots, \text{pat}_n] \Rightarrow (\text{pat}), StE} \quad (411)$$

$$\frac{C \vdash \text{pat} \Rightarrow \text{pat}', StE}{C \vdash (\text{pat}) \Rightarrow (\text{pat}'), StE} \quad (412)$$

Comments:

(411) It is assumed here that neither `nil` nor `::` has been rebound, and `::` still has right-associative infix status.

Pattern Rows

$$\boxed{C \vdash \text{patrow} \Rightarrow \text{patrow}', StE}$$

$$\overline{C \vdash \dots \Rightarrow \dots, \{ \}} \quad (413)$$

$$\frac{C \vdash \text{pat} \Rightarrow \text{pat}', StE \quad \langle C \vdash \text{patrow} \Rightarrow \text{patrow}', StE' \rangle}{C \vdash \text{lab} = \text{pat} \langle _, \text{patrow} \rangle \Rightarrow \text{lab} = \text{pat}' \langle _, \text{patrow}' \rangle, StE \langle + StE' \rangle} \quad (414)$$

$$\frac{\text{lab} = \text{id} \text{ in Lab} \quad C \vdash \text{lab} = \text{id} \langle : \text{ty} \rangle \langle \langle _, \text{patrow} \rangle \rangle \Rightarrow \text{patrow}', StE}{C \vdash \text{id} \langle : \text{ty} \rangle \langle \langle _, \text{patrow} \rangle \rangle \Rightarrow \text{patrow}', StE} \quad (415)$$

$$\frac{\text{lab} = \text{id} \text{ in Lab} \quad C \vdash \text{lab} = \text{id} \langle : \text{ty} \rangle \text{ as pat} \langle \langle _, \text{patrow} \rangle \rangle \Rightarrow \text{patrow}', StE}{C \vdash \text{id} \langle : \text{ty} \rangle \text{ as pat} \langle \langle _, \text{patrow} \rangle \rangle \Rightarrow \text{patrow}', StE} \quad (416)$$

Application Patterns

$$\boxed{C \vdash \text{apppat} \Rightarrow \text{pat}, \text{StE}}$$

$$\frac{C \vdash \langle \text{op} \rangle \text{longid} \Rightarrow \text{longid}', st \quad st \neq \mathbf{v} \quad C \vdash \text{atpat} \Rightarrow \text{atpat}', \text{StE}}{C \vdash \langle \text{op} \rangle \text{longid} \text{ atpat} \Rightarrow \text{longid}' \text{ atpat}', \text{StE}} \quad (417)$$

$$\frac{C \vdash \text{atpat} \Rightarrow \text{atpat}', \text{StE}}{C \vdash \text{atpat} \Rightarrow \text{atpat}', \text{StE}} \quad (418)$$

Infix Patterns

$$\boxed{C, n, \text{fix} \vdash \text{infp}at \Rightarrow \text{pat}, \text{StE}}$$

$$\frac{C \vdash \text{apppat} \Rightarrow \text{pat}, \text{StE}}{C, n, \text{fix} \vdash \text{apppat} \Rightarrow \text{pat}, \text{StE}} \quad (419)$$

$$\frac{\begin{array}{l} [\text{VE of } C](id) = (n', 1) \quad n < n' \vee (n = n' \wedge \text{fix} = 1) \\ [\text{StE of } E \text{ of } C](id) = st \in \{\mathbf{c}, \mathbf{e}\} \\ C, n', 1 \vdash \text{infp}at_1 \Rightarrow \text{pat}_1, \text{StE}_1 \quad C, n', \mathbf{n} \vdash \text{infp}at_2 \Rightarrow \text{pat}_2, \text{StE}_2 \end{array}}{C, n, \text{fix} \vdash \text{infp}at_1 \text{ id } \text{infp}at_2 \Rightarrow id^{st} \{1=\text{pat}_1, 2=\text{pat}_2\}, \text{StE}_1 + \text{StE}_2} \quad (420)$$

$$\frac{\begin{array}{l} [\text{VE of } C](id) = (n', \mathbf{r}) \quad n < n' \vee (n = n' \wedge \text{fix} = \mathbf{r}) \\ [\text{StE of } E \text{ of } C](id) = st \in \{\mathbf{c}, \mathbf{e}\} \\ C, n', \mathbf{n} \vdash \text{infp}at_1 \Rightarrow \text{pat}_1, \text{StE}_1 \quad C, n', \mathbf{r} \vdash \text{infp}at_2 \Rightarrow \text{pat}_2, \text{StE}_2 \end{array}}{C, n, \text{fix} \vdash \text{infp}at_1 \text{ id } \text{infp}at_2 \Rightarrow id^{st} \{1=\text{pat}_1, 2=\text{pat}_2\}, \text{StE}_1 + \text{StE}_2} \quad (421)$$

Patterns

$$\boxed{C \vdash \text{pat} \Rightarrow \text{pat}', \text{StE}}$$

$$\frac{C, 0, \text{fix} \vdash \text{infp}at \Rightarrow \text{pat}, \text{StE}}{C \vdash \text{infp}at \Rightarrow \text{pat}, \text{StE}} \quad (422)$$

$$\frac{C \vdash \text{ty} \Rightarrow \text{ty}' \quad C \vdash \text{pat} \Rightarrow \text{pat}', \text{StE}}{C \vdash \text{pat} : \text{ty} \Rightarrow \text{pat}' : \text{ty}', \text{StE}} \quad (423)$$

$$\frac{C \vdash \langle \text{op} \rangle id \Rightarrow \text{var}, \mathbf{v} \quad \langle\langle C \vdash \text{ty} \Rightarrow \text{ty}' \rangle\rangle \quad C \vdash \text{pat} \Rightarrow \text{pat}', \text{StE}}{C \vdash \langle \text{op} \rangle id \langle\langle : \text{ty} \rangle\rangle \text{ as } \text{pat} \Rightarrow \text{var} \langle\langle : \text{ty}' \rangle\rangle \text{ as } \text{pat}', \{id \mapsto \mathbf{v}\} + \text{StE}} \quad (424)$$

Function Patterns

$$\boxed{C \vdash \text{fpat} \Rightarrow \text{pat}, \text{var}, n, \text{StE}}$$

$$\frac{n \geq 1 \quad C \vdash (\text{atpat}_1, \dots, \text{atpat}_n) \Rightarrow \text{pat}, \text{StE}}{C \vdash \text{op } id \text{ atpat}_1 \dots \text{atpat}_n \Rightarrow \text{pat}, id^{\mathbf{V}}, n, \text{StE}} \quad (425)$$

$$\frac{n \geq 1 \quad [\text{VE of } C](id) = (n', \mathbf{n}) \quad C \vdash (\text{atpat}_1, \dots, \text{atpat}_n) \Rightarrow \text{pat}, \text{StE}}{C \vdash id \text{ atpat}_1 \dots \text{atpat}_n \Rightarrow \text{pat}, id^{\mathbf{V}}, n, \text{StE}} \quad (426)$$

$$\frac{C \vdash ((atpat_1, atpat'_1), atpat_2, \dots, atpat_n) \Rightarrow pat, StE \quad [VE \text{ of } C](id) = (n', fix) \quad fix \neq \mathbf{n} \quad n \geq 1}{C \vdash (atpat_1 \text{ id } atpat'_1) atpat_2 \dots atpat_n \Rightarrow pat, id^{\mathbf{V}}, n, StE} \quad (427)$$

$$\frac{[VE \text{ of } C](id) = (n', fix) \quad fix \neq \mathbf{n} \quad C \vdash (atpat_1, atpat'_1) \Rightarrow pat, StE}{C \vdash atpat_1 \text{ id } atpat'_1 \Rightarrow pat, id^{\mathbf{V}}, 1, StE} \quad (428)$$

Type Expressions

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

$$\frac{tyseq = ty_1 \dots ty_k \quad C \vdash ty_1 \Rightarrow ty'_1 \quad \dots \quad C \vdash ty_k \Rightarrow ty'_k \quad C, TE \text{ of } E \text{ of } _, \mathbf{t} \vdash longid \Rightarrow longtycon, \Lambda \alpha_1 \dots \alpha_k. ty}{C \vdash tyseq \text{ longid} \Rightarrow ty\{ty'_1/\alpha_1, \dots, ty'_k/\alpha_k\}} \quad (429)$$

$$\frac{C, TE \text{ of } E \text{ of } _, \mathbf{t} \vdash longid \Rightarrow longtycon, FAIL \quad tyseq = ty_1 \dots ty_k \quad C \vdash ty_1 \Rightarrow ty'_1 \quad \dots \quad C \vdash ty_k \Rightarrow ty'_k \quad tyseq' = ty'_1 \dots ty'_k}{C \vdash tyseq \text{ longid} \Rightarrow tyseq' \text{ longtycon}} \quad (430)$$

$$\frac{n \geq 2 \quad C \vdash \{1: ty_1, \dots, \bar{n}: ty_n\} \Rightarrow ty}{C \vdash ty_1 * \dots * ty_n \Rightarrow ty} \quad (431)$$

Comments:

(429) 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

$$\boxed{C \vdash strexp \Rightarrow strexp', E}$$

$$\frac{C \vdash strdec \Rightarrow strdec', C'}{C \vdash \mathbf{struct} \text{ strdec } \mathbf{end} \Rightarrow \mathbf{struct} \text{ strdec}' \mathbf{end}, E \text{ of } C'} \quad (432)$$

$$\frac{C, SE \text{ of } E \text{ of } _, \mathbf{s} \vdash longid \Rightarrow longstrid, E}{C \vdash longid \Rightarrow longstrid, E} \quad (433)$$

$$\frac{C, F \text{ of } _, \mathbf{f} \vdash id \Rightarrow funid, (E_1, E_2) \quad C \vdash strexp \Rightarrow strexp', E \quad E \succ E_1}{C \vdash id (\text{strexp}) \vdash funid (\text{strexp}'), E_2} \quad (434)$$

$$\frac{C \vdash id (\mathbf{struct} \text{ strdec } \mathbf{end}) \Rightarrow strexp', E}{C \vdash id (\text{strdec}) \Rightarrow strexp', E} \quad (435)$$

$$\frac{C \vdash strdec \Rightarrow strdec', C' \quad C + C' \vdash strexp \Rightarrow strexp', E}{C \vdash \mathbf{let} \text{ strdec } \mathbf{in} \text{ strexp } \mathbf{end} \Rightarrow \mathbf{let} \text{ strdec}' \mathbf{in} \text{ strexp}' \mathbf{end}, E} \quad (436)$$

Structure-level Declarations

$$\boxed{C \vdash \text{strdec} \Rightarrow \text{strdec}', C'}$$

$$\frac{C \vdash \text{dec} \Rightarrow \text{dec}', C'}{C \vdash \text{dec} \Rightarrow \text{dec}', C'} \quad (437)$$

$$\frac{C \vdash \text{ax} \Rightarrow \text{ax}'}{C \vdash \text{axiom } \text{ax} \Rightarrow \text{axiom } \text{ax}', \{\} \text{ in Context}} \quad (438)$$

$$\frac{C \vdash \text{strbind} \Rightarrow \text{strbind}', SE}{C \vdash \text{structure } \text{strbind} \Rightarrow \text{structure } \text{strbind}', SE \text{ in Env in Context}} \quad (439)$$

$$\frac{C \vdash \text{strdec}_1 \Rightarrow \text{strdec}'_1, C_1 \quad C + C_1 \vdash \text{strdec}_2 \Rightarrow \text{strdec}'_2, C_2}{C \vdash \text{local } \text{strdec}_1 \text{ in } \text{strdec}_2 \text{ end} \Rightarrow \text{local } \text{strdec}'_1 \text{ in } \text{strdec}'_2 \text{ end}, C_2} \quad (440)$$

$$\overline{C \vdash \quad \Rightarrow \quad, \{\} \text{ in Context}} \quad (441)$$

$$\frac{C \vdash \text{strdec}_1 \Rightarrow \text{strdec}'_1, C_1 \quad C + C_1 \vdash \text{strdec}_2 \Rightarrow \text{strdec}'_2, C_2}{C \vdash \text{strdec}_1 \langle ; \rangle \text{strdec}_2 \Rightarrow \text{strdec}'_1 \langle ; \rangle \text{strdec}'_2, C_1 + C_2} \quad (442)$$

Structure Bindings

$$\boxed{C \vdash \text{strbind} \Rightarrow SE}$$

$$\frac{C \vdash \text{sglstrbind} \Rightarrow \text{sglstrbind}', SE_1 \quad \langle C \vdash \text{strbind} \Rightarrow \text{strbind}', SE_2 \rangle}{C \vdash \text{sglstrbind} \langle \text{and strbind} \rangle \Rightarrow \text{sglstrbind}' \langle \text{and strbind}' \rangle, SE_1 \langle +SE_2 \rangle} \quad (443)$$

Single Structure Bindings

$$\boxed{C \vdash \text{sglstrbind} \Rightarrow \text{sglstrbind}', SE}$$

$$\frac{C \vdash \text{psigexp} \Rightarrow \text{psigexp}', E_1 \quad C \vdash \text{strex} \Rightarrow \text{strex}', E_2 \quad E_2 \succ E_1}{C \vdash \text{id} : \text{psigexp} = \text{strex} \Rightarrow \text{id}^{\mathbf{S}} : \text{psigexp}' = \text{strex}', \{\text{id} \mapsto E_1\}} \quad (444)$$

$$\frac{C \vdash \text{psigexp} \Rightarrow \text{psigexp}', E}{C \vdash \text{id} : \text{psigexp} = ? \Rightarrow \text{id}^{\mathbf{S}} : \text{psigexp}' = ?, \{\text{id} \mapsto E\}} \quad (445)$$

$$\frac{C \vdash \text{strex} \Rightarrow \text{strex}', E}{C \vdash \text{id} = \text{strex} \Rightarrow \text{id}^{\mathbf{S}} = \text{strex}', \{\text{id} \mapsto E\}} \quad (446)$$

Signature Expressions

$$\boxed{C \vdash \text{sigexp} \Rightarrow \text{sigexp}', E}$$

$$\frac{C \vdash \text{spec} \Rightarrow \text{spec}', E}{C \vdash \text{sig } \text{spec} \text{ end} \Rightarrow \text{sig } \text{spec}' \text{ end}, E} \quad (447)$$

$$\frac{C, G \text{ of } _, \mathbf{g} \vdash \text{id} \Rightarrow \text{sigid}, E}{C \vdash \text{id} \Rightarrow \text{sigid}, E} \quad (448)$$

Principal Signatures

$$\boxed{C \vdash \text{psigexp} \Rightarrow \text{psigexp}', E}$$

$$\frac{C \vdash \text{sigexp} \Rightarrow \text{sigexp}', E}{C \vdash \text{sigexp} \Rightarrow \text{sigexp}', E} \quad (449)$$

Signature Declarations

$$\boxed{C \vdash \text{sigdec} \Rightarrow \text{sigdec}', G}$$

$$\frac{C \vdash \text{sigbind} \Rightarrow \text{sigbind}', G}{C \vdash \text{signature sigbind} \Rightarrow \text{signature sigbind}', G} \quad (450)$$

$$\overline{C \vdash \Rightarrow \quad , \{ \}} \quad (451)$$

$$\frac{C \vdash \text{sigdec}_1 \Rightarrow \text{sigdec}'_1, G_1 \quad C + G_1 \vdash \text{sigdec}_2 \Rightarrow \text{sigdec}'_2, G_2}{C \vdash \text{sigdec}_1 \langle ; \rangle \text{sigdec}_2 \Rightarrow \text{sigdec}'_1 \langle ; \rangle \text{sigdec}'_2, G_1 + G_2} \quad (452)$$

Signature Bindings

$$\boxed{C \vdash \text{sigbind} \Rightarrow \text{sigbind}', G}$$

$$\frac{C \vdash \text{psigexp} \Rightarrow \text{psigexp}', E \quad \langle C \vdash \text{sigbind} \Rightarrow \text{sigbind}', G \rangle}{C \vdash \text{id} = \text{psigexp} \langle \text{and sigbind} \rangle \Rightarrow \text{id}^{\mathcal{G}} = \text{psigexp}' \langle \text{and sigbind}' \rangle, \{ \text{id} \mapsto E \} \langle +G \rangle} \quad (453)$$

Specifications

$$\boxed{C \vdash \text{spec} \Rightarrow \text{spec}', E}$$

$$\frac{C \vdash \text{valdesc} \Rightarrow \text{valdesc}', E}{C \vdash \text{val valdesc} \Rightarrow \text{val valdesc}', E} \quad (454)$$

$$\frac{C \vdash \text{typdesc} \Rightarrow \text{typdesc}', TE}{C \vdash \text{type typdesc} \Rightarrow \text{type typdesc}', TE \text{ in Env}} \quad (455)$$

$$\frac{C \vdash \text{typdesc} \Rightarrow \text{typdesc}', TE}{C \vdash \text{eqtype typdesc} \Rightarrow \text{eqtype typdesc}', TE \text{ in Env}} \quad (456)$$

$$\frac{C \vdash \text{datdesc} \Rightarrow \text{datdesc}', E}{C \vdash \text{datatype datdesc} \Rightarrow \text{datatype datdesc}', E} \quad (457)$$

$$\frac{C \vdash \text{exdesc} \Rightarrow \text{exdesc}', StE}{C \vdash \text{exception exdesc} \Rightarrow \text{exception exdesc}', StE \text{ in Env}} \quad (458)$$

$$\frac{C \vdash \text{axdesc} \Rightarrow \text{axdesc}'}{C \vdash \text{axiom axdesc} \Rightarrow \text{axiom axdesc}', \{ \} \text{ in Env}} \quad (459)$$

$$\frac{C \vdash \text{strdesc} \Rightarrow \text{strdesc}', SE}{C \vdash \text{structure strdesc} \Rightarrow \text{structure strdesc}', SE \text{ in Env}} \quad (460)$$

$$\frac{C \vdash \text{shareq} \Rightarrow \text{shareq}'}{C \vdash \text{sharing } \text{shareq} \Rightarrow \text{sharing } \text{shareq}', \{\} \text{ in Env}} \quad (461)$$

$$\frac{C \vdash \text{spec}_1 \Rightarrow \text{spec}'_1, E_1 \quad C + E_1 \vdash \text{spec}_2 \Rightarrow \text{spec}'_2, E_2}{C \vdash \text{local } \text{spec}_1 \text{ in } \text{spec}_2 \text{ end} \Rightarrow \text{local } \text{spec}'_1 \text{ in } \text{spec}'_2 \text{ end}, E_2} \quad (462)$$

$$\frac{\begin{array}{c} C, SE \text{ of } E \text{ of } _, \mathbf{s} \vdash \text{longid}_1 \Rightarrow \text{longstrid}_1, E_1 \\ \dots \\ C, SE \text{ of } E \text{ of } _, \mathbf{s} \vdash \text{longid}_n \Rightarrow \text{longstrid}_n, E_n \end{array}}{C \vdash \text{open } \text{longid}_1 \dots \text{longid}_n \Rightarrow \text{open } \text{longstrid}_1 \dots \text{longstrid}_n, E_1 + \dots + E_n} \quad (463)$$

$$\frac{\begin{array}{c} C, G \text{ of } _, \mathbf{g} \vdash \text{id}_1 \Rightarrow \text{sigid}_1, E_1 \\ \dots \\ C, G \text{ of } _, \mathbf{g} \vdash \text{id}_n \Rightarrow \text{sigid}_n, E_n \end{array}}{C \vdash \text{include } \text{id}_1 \dots \text{id}_n \Rightarrow \text{include } \text{sigid}_1 \dots \text{sigid}_n, E_1 + \dots + E_n} \quad (464)$$

$$\frac{}{C \vdash \quad \Rightarrow \quad, \{\} \text{ in Env}} \quad (465)$$

$$\frac{C \vdash \text{spec}_1 \Rightarrow \text{spec}'_1, E_1 \quad C + E_1 \vdash \text{spec}_2 \Rightarrow \text{spec}'_2, E_2}{C \vdash \text{spec}_1 \langle ; \rangle \text{spec}_2 \Rightarrow \text{spec}'_1 \langle ; \rangle \text{spec}'_2, E_1 + E_2} \quad (466)$$

Value Descriptions

$$\boxed{C \vdash \text{valdesc} \Rightarrow \text{valdesc}', StE}$$

$$\frac{C \vdash \text{ty} \Rightarrow \text{ty}' \quad \langle C \vdash \text{valdesc} \Rightarrow \text{valdesc}', StE \rangle}{C \vdash \text{id} : \text{ty} \langle \text{and } \text{valdesc} \rangle \Rightarrow \text{id}^{\mathbf{v}} : \text{ty}' \langle \text{and } \text{valdesc}' \rangle, \{\text{id} \mapsto \mathbf{v}\} \langle +StE \rangle} \quad (467)$$

Type Descriptions

$$\boxed{C \vdash \text{typdesc} \Rightarrow \text{typdesc}', TE}$$

$$\frac{\langle C \vdash \text{typdesc} \Rightarrow \text{typdesc}', TE \rangle}{C \vdash \text{tyvarseq } \text{id} \langle \text{and } \text{typdesc} \rangle \Rightarrow \text{tyvarseq } \text{id}^{\mathbf{t}} \langle \text{and } \text{typdesc}' \rangle, \{\text{id} \mapsto \text{FAIL}\} \langle +TE \rangle} \quad (468)$$

Datatype Description

$$\boxed{C \vdash \text{datdesc} \Rightarrow \text{datdesc}', E}$$

$$\frac{C \vdash \text{condesc} \Rightarrow \text{condesc}', StE \quad \langle C \vdash \text{datdesc} \Rightarrow \text{datdesc}', E \rangle}{C \vdash \text{tyvarseq } \text{id} = \text{condesc} \langle \text{and } \text{datdesc} \rangle \Rightarrow \text{tyvarseq } \text{id}^{\mathbf{t}} = \text{condesc}' \langle \text{and } \text{datdesc}' \rangle, (\{\text{id} \mapsto \text{FAIL}\}, StE) \text{ in Env } \langle +E \rangle} \quad (469)$$

Constructor Descriptions

$$\boxed{C \vdash \text{condesc} \Rightarrow \text{condesc}', StE}$$

$$\frac{\langle C \vdash ty \Rightarrow ty' \rangle \quad \langle \langle C \vdash \text{condesc} \Rightarrow \text{condesc}', StE \rangle \rangle}{C \vdash id \langle \text{of } ty \rangle \langle \langle | \text{condesc} \rangle \rangle \Rightarrow id^c \langle \text{of } ty' \rangle \langle \langle | \text{condesc}' \rangle \rangle, \{id \mapsto c\} \langle \langle +StE \rangle \rangle} \quad (470)$$

Exception Descriptions

$$\boxed{C \vdash \text{exdesc} \Rightarrow \text{exdesc}', StE}$$

$$\frac{\langle C \vdash ty \Rightarrow ty' \rangle \quad \langle \langle C \vdash \text{exdesc} \Rightarrow \text{exdesc}', StE \rangle \rangle}{C \vdash id \langle \text{of } ty \rangle \langle \langle \text{and exdesc} \rangle \rangle \Rightarrow id^e \langle \text{of } ty' \rangle \langle \langle \text{and exdesc}' \rangle \rangle, \{id \mapsto e\} \langle \langle +StE \rangle \rangle} \quad (471)$$

Specification Expressions

$$\boxed{C \vdash \text{specexp} \Rightarrow \text{specexp}'}$$

$$\frac{C \vdash \text{strdec} \Rightarrow \text{strdec}', E \quad C + E \vdash \text{axexp} \Rightarrow \text{exp}}{C \vdash \text{let } \text{strdec} \text{ in } \text{axexp} \text{ end} \Rightarrow \text{let } \text{strdec}' \text{ in } \text{exp} \text{ end}} \quad (472)$$

$$\frac{C \vdash \text{let} \quad \text{in } \text{axexp} \text{ end} \Rightarrow \text{specexp}}{C \vdash \text{axexp} \Rightarrow \text{specexp}} \quad (473)$$

Axiomatic Expressions

$$\boxed{C \vdash \text{axexp} \Rightarrow \text{exp}}$$

$$\frac{C \vdash \text{exp}^\bullet \Rightarrow \text{exp}}{C \vdash \text{exp}^\bullet \Rightarrow \text{exp}} \quad (474)$$

Structure Descriptions

$$\boxed{C \vdash \text{strdesc} \Rightarrow \text{strdesc}', SE}$$

$$\frac{C \vdash \text{sigexp} \Rightarrow \text{sigexp}', E \quad \langle C \vdash \text{strdesc} \Rightarrow \text{strdesc}', SE \rangle}{C \vdash id : \text{sigexp} \langle \text{and strdesc} \rangle \Rightarrow id^s : \text{sigexp}' \langle \text{and strdesc}' \rangle, \{id \mapsto E\} \langle +SE \rangle} \quad (475)$$

Functor Declarations

$$\boxed{C \vdash \text{fundec} \Rightarrow \text{fundec}', F}$$

$$\frac{C \vdash \text{funbind} \Rightarrow \text{funbind}', F}{C \vdash \text{functor } \text{funbind} \Rightarrow \text{functor } \text{funbind}', F} \quad (476)$$

$$\overline{C \vdash \quad \Rightarrow \quad , \{ \}} \quad (477)$$

$$\frac{C \vdash \text{fundec}_1 \Rightarrow \text{fundec}'_1, F_1 \quad C + F_1 \vdash \text{fundec}_2 \Rightarrow \text{fundec}'_2, F_2}{C \vdash \text{fundec}_1 \langle ; \rangle \text{fundec}_2 \Rightarrow \text{fundec}'_1 \langle ; \rangle \text{fundec}'_2, F_1 + F_2} \quad (478)$$

Functor Bindings

$$\boxed{C \vdash \text{funbind} \Rightarrow \text{funbind}', F}$$

$$\frac{\begin{array}{l} C, SE \text{ of } E \text{ of } _, s \vdash id'' \Rightarrow \text{strid}, \text{FAIL} \\ C \vdash id (id'' : \text{sig spec end}) : id' = \\ \quad \text{let open } id'' \text{ in } \text{strexpr end} \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F \end{array}}{C \vdash id (\text{spec}) : id' = \text{strexpr} \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F} \quad (479)$$

$$\frac{\begin{array}{l} C, SE \text{ of } E \text{ of } _, s \vdash id' \Rightarrow \text{strid}, \text{FAIL} \\ C \vdash id (id' : \text{sig spec end}) : \text{sig local open } id' \text{ in } \text{spec}' \text{ end end} \\ \quad = \text{let open } id' \text{ in } \text{strexpr end} \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F \end{array}}{C \vdash id (\text{spec}) : \text{sig spec}' \text{ end} = \text{strexpr} \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F} \quad (480)$$

$$\frac{\begin{array}{l} C, SE \text{ of } E \text{ of } _, s \vdash id'' \Rightarrow \text{strid}, \text{FAIL} \\ C \vdash id (id'' : \text{sig spec end}) : id' = ? \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F \end{array}}{C \vdash id (\text{spec}) : id' = ? \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F} \quad (481)$$

$$\frac{\begin{array}{l} C, SE \text{ of } E \text{ of } _, s \vdash id' \Rightarrow \text{strid}, \text{FAIL} \\ C \vdash id (id' : \text{sig spec end}) : \text{sig local open } id' \text{ in } \text{spec}' \text{ end end} \\ \quad = ? \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F \end{array}}{C \vdash id (\text{spec}) : \text{sig spec}' \text{ end} = ? \langle \text{and funbind} \rangle \Rightarrow \text{funbind}', F} \quad (482)$$

$$\frac{\begin{array}{l} C \vdash \text{psigexp}_1 \Rightarrow \text{psigexp}'_1, E_1 \\ C + (\{id_2 \mapsto E_1\} \text{ in Env}) \vdash \text{psigexp}_2 \Rightarrow \text{psigexp}'_2, E_2 \\ C + (\{id_2 \mapsto E_1\} \text{ in Env}) \vdash \text{strexpr} \Rightarrow \text{strexpr}', E \quad E \succ E_2 \\ \langle C \vdash \text{funbind} \Rightarrow \text{funbind}', F \rangle \end{array}}{C \vdash id_1 (id_2 : \text{psigexp}_1) : \text{psigexp}_2 = \text{strexpr} \langle \text{and funbind} \rangle \Rightarrow \\ id_1^{\mathbf{f}} (id_2^{\mathbf{S}} : \text{psigexp}'_1) : \text{psigexp}'_2 = \text{strexpr}' \langle \text{and funbind}' \rangle, \\ \{ \text{funid} \mapsto (E_1, E_2) \} \langle +F \rangle} \quad (483)$$

$$\frac{\begin{array}{l} C \vdash \text{psigexp}_1 \Rightarrow \text{psigexp}'_1, E_1 \\ C + (\{id_2 \mapsto E_1\} \text{ in Env}) \vdash \text{psigexp}_2 \Rightarrow \text{psigexp}'_2, E_2 \\ \langle C \vdash \text{funbind} \Rightarrow \text{funbind}', F \rangle \end{array}}{C \vdash id_1 (id_2 : \text{psigexp}_1) : \text{psigexp}_2 = ? \langle \text{and funbind} \rangle \Rightarrow \\ id_1^{\mathbf{f}} (id_2^{\mathbf{S}} : \text{psigexp}'_1) : \text{psigexp}'_2 = ? \langle \text{and funbind}' \rangle, \\ \{ \text{funid} \mapsto (E_1, E_2) \} \langle +F \rangle} \quad (484)$$

Top-level Declarations

$$\boxed{C \vdash \text{topdec} \Rightarrow \text{topdec}', C'}$$

$$\frac{C \vdash \text{val it} = \text{exp} \Rightarrow \text{topdec}, C'}{C \vdash \text{exp} \Rightarrow \text{topdec}, C'} \quad (485)$$

$$\frac{C \vdash \text{strdec} \Rightarrow \text{strdec}', C'}{C \vdash \text{strdec} \Rightarrow \text{strdec}', C'} \quad (486)$$

$$\frac{C \vdash \text{sigdec} \Rightarrow \text{sigdec}', G}{C \vdash \text{sigdec} \Rightarrow \text{sigdec}', G \text{ in Context}} \quad (487)$$

$$\frac{C \vdash \text{fundec} \Rightarrow \text{fundec}', F}{C \vdash \text{fundec} \Rightarrow \text{fundec}', F \text{ in Context}} \quad (488)$$

C Appendix: The Initial Basis

For the verification of full programs, see rule 349, we have to specify the initial (verification) basis B_0 , an initial context C_{DER}^0 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

$$s_0 = (\text{BasExName}, \{\}, \{\}, CT_0 \cup ET_0, 0, \top)$$

where BasExName can be found in section 6.5 on page 58. 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 304, which allows an arbitrary choice of question mark interpretation.

The component l of s_0 is set to \top . This means that structure declarations ignore the actual “implementing” structure.

$CT_0 = \{ (\text{true}, \text{bool}), (\text{false}, \text{bool}), (\text{nil}, \forall 'a. 'a \text{ list}), \\ (\text{:}, \forall \alpha. \{1 \mapsto 'a, 2 \mapsto 'a \text{ list}\} \rightarrow 'a \text{ list}) \}$ $ET_0 = \{ (\text{excon}, \text{exn}) \mid \text{excon} \in \text{BasExName} \setminus \{ \text{NoCode}, \text{Abuse} \} \}$
--

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_{DER}^0 has the form

$$C_{\text{DER}}^0 = ((SE_{\text{DER}}, TE_{\text{DER}}, StE^0), VE_{\text{DER}}^0, F_{\text{DER}}, SE_{\text{DER}})$$

where all components except VE_{DER}^0 and StE^0 are the empty map $\{\}$. VE_{DER}^0 is defined in figure 28, StE^0 in figure 29. The set BasVal is defined in section 6.4, page 58. PredFun is the set containing the identifiers

o @ ^ map rev not

id	\mapsto	(n, fix)	$condition$
o	\mapsto	$(3, 1)$	
id	\mapsto	$(4, 1)$	$id \in \{=, <>, <, >, <=, >=\}$
$@$	\mapsto	$(5, 1)$	
$::$	\mapsto	$(5, r)$	
id	\mapsto	$(6, 1)$	$id \in \{+, -, \wedge\}$
id	\mapsto	$(7, 1)$	$id \in \{\text{div}, \text{mod}, /, *\}$

Figure 28: Initial Infix Status VE_{DER}^0

id	\mapsto	st	$condition$
id	\mapsto	e	$id \in \text{BasExName} \setminus \{\text{NoCode}\}$
id	\mapsto	c	$id \in \{\text{true}, \text{false}, ::, \text{nil}\}$
id	\mapsto	v	$id \in \text{BasVal} \cup \text{PredFun}$

Figure 29: Initial Status Environment StE^0

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.

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

- $M_0 = \emptyset$
- $T_0 = \{\text{bool}, \text{int}, \text{real}, \text{string}, \text{list}, \text{exn}\}$
- $F_0 = \{\}$
- $G_0 = \{\}$
- $E_0 = (SE_0, TE_0, VE_0)$
- $SE_0 = \{\}$

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 $\text{Dom } VE'_0 = \text{BasVal} \cup \{\text{true}, \text{false}, ::, \text{nil}\} \cup (\text{BasExName} \setminus \{\text{NoCode}, \text{Abuse}\})$ and $\text{Dom } VE''_0 = \text{PredFun}$. All identifiers id in the domain of VE'_0 are mapped to typed values tv . The type schemes $\text{Stat } tv$ of the predefined identifiers are given in figure 31; the basic exceptions (not listed there) have all type `exn`.

<i>tycon</i>	\mapsto	$\{ \theta, \quad \{ \text{con}_1 \mapsto \sigma_1, \dots, \text{con}_n \mapsto \sigma_n \} \quad (n \geq 0) \}$
<code>unit</code>	\mapsto	$\{ \Lambda().\{\}, \quad \{\} \}$
<code>bool</code>	\mapsto	$\{ \text{bool}, \quad \{ \text{true} \mapsto \text{bool}, \text{false} \mapsto \text{bool} \} \}$
<code>int</code>	\mapsto	$\{ \text{int}, \quad \{\} \}$
<code>real</code>	\mapsto	$\{ \text{real}, \quad \{\} \}$
<code>string</code>	\mapsto	$\{ \text{string}, \quad \{\} \}$
<code>list</code>	\mapsto	$\{ \text{list}, \quad \{ \text{nil} \mapsto \forall 'a . 'a \text{ list}, \quad :: \mapsto \forall 'a . \{ 1 \mapsto 'a, 2 \mapsto 'a \text{ list} \} \rightarrow 'a \text{ list} \} \}$
<code>exn</code>	\mapsto	$\{ \text{exn}, \quad \{\} \}$

Figure 30: Static TE_0

NONFIX	INFIX
$var \mapsto \sigma$	$var \mapsto \sigma$
map $\mapsto \forall 'a 'b. ('a \rightarrow 'b) \rightarrow$ $'a \text{ list} \rightarrow 'b \text{ list}$	Precedence 7 : / $\mapsto \text{real} * \text{real} \rightarrow \text{real}$
rev $\mapsto \forall 'a. 'a \text{ list} \rightarrow 'a \text{ list}$	div $\mapsto \text{int} * \text{int} \rightarrow \text{int}$
not $\mapsto \text{bool} \rightarrow \text{bool}$	mod $\mapsto \text{int} * \text{int} \rightarrow \text{int}$
~ $\mapsto \forall \text{num}. \text{num} \rightarrow \text{num}$	* $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{num}$
abs $\mapsto \forall \text{num}. \text{num} \rightarrow \text{num}$	Precedence 6 :
floor $\mapsto \text{real} \rightarrow \text{int}$	+ $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{num}$
real $\mapsto \text{int} \rightarrow \text{real}$	- $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{num}$
sqrt $\mapsto \text{real} \rightarrow \text{real}$	^ $\mapsto \text{string} * \text{string} \rightarrow \text{string}$
sin $\mapsto \text{real} \rightarrow \text{real}$	Precedence 5 :
cos $\mapsto \text{real} \rightarrow \text{real}$:: $\mapsto \forall 'a. 'a * 'a \text{ list} \rightarrow 'a \text{ list}$
arctan $\mapsto \text{real} \rightarrow \text{real}$	@ $\mapsto \forall 'a. 'a \text{ list}$ $* 'a \text{ list} \rightarrow 'a \text{ list}$
exp $\mapsto \text{real} \rightarrow \text{real}$	Precedence 4 :
ln $\mapsto \text{real} \rightarrow \text{real}$	= $\mapsto \forall 'a. ''a * ''a \rightarrow \text{bool}$
size $\mapsto \text{string} \rightarrow \text{int}$	<> $\mapsto \forall 'a. ''a * ''a \rightarrow \text{bool}$
chr $\mapsto \text{int} \rightarrow \text{string}$	< $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{bool}$
ord $\mapsto \text{string} \rightarrow \text{int}$	> $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{bool}$
explode $\mapsto \text{string} \rightarrow \text{string list}$	<= $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{bool}$
implode $\mapsto \text{string list} \rightarrow \text{string}$	>= $\mapsto \forall \text{num}. \text{num} * \text{num} \rightarrow \text{bool}$
true $\mapsto \text{bool}$	Precedence 3 :
false $\mapsto \text{bool}$	o $\mapsto \forall 'a 'b 'c. ('b \rightarrow 'c)$ $* ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'c)$
nil $\mapsto \forall 'a. 'a \text{ list}$	

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 $\text{Stat } VE_0$ (without exceptions)

C.4 Predefined Functions

Note that $\text{Dyn } VE'_0$ 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 verified with the program semantics (page 117) in the initial state and basis as $s_0, B'_0, C'_{\text{DER}} \vdash \text{topdec} \Rightarrow \mathcal{B}, C'_{\text{DER}}$ 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 $b \in \text{BasVal}$. 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 ↦ i, 2 ↦ d})`” to “ $i \bmod d$ ”, etc.

- $\sim(n)$ 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.
- `v1 = v2` 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.
- `v1 <> v2` returns the opposite boolean value to `v1 = v2`.

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).