A Type-Theoretic Interpretation of Standard ML*

Robert Harper and Chris Stone
{rwh,cstone}@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
Draft submitted for publication, May 1997

1 Introduction

It has been nearly twenty years since Robin Milner introduced ML as the metalanguage of the LCF interactive theorem prover [5]. The design of ML has proved to be a decisive step in the development of programming languages. Milner's elegant use of abstract types to ensure validity of machine-generated proofs, combined with his innovative and flexible polymorphic type discipline, and supported by a rigorous proof of soundness of the language provided a convincing argument for the importance of type systems for programming languages. Today ML serves as the standard against which languages are measured.

Milner's seminal work on ML has given rise to a large body of research on type systems for programming languages.[1] Type theory provides a uniform conceptual framework that gives substance to informal ideas such as "orthogonality" and "safety" and provides a test bed for evaluating and comparing languages. Type theory has also begun to be exploited as an implementation tool [26, 25]. By exploiting type information at compile-, link-, and run-time an implementation may use more efficient data representations than was previously possible.

Milner's work on ML culminated with his ambitious proposal for Standard ML [16] which sought to extend ML to a full-scale programming language supporting functional and imperative programming and an expressive module system. Standard ML presented a serious challenge to rigorous formalization of its static and dynamic semantics. A key difficulty for static semantics is the management of the propagation of type information in a program. This is essential for enforcing abstraction without imposing undue notational burdens on the programmer.

The first successful formalization of Standard ML was given in *The Definition of Standard ML*. *The Definition* introduced a number of novel ideas, including using a relational framework to specify both the static and dynamic semantics, and maintaining a clear separation between definitional and algorithmic considerations. The management of type sharing information was accomplished through the use of "generative stamps", according to which abstract types are assigned unique stamps (or "names") to ensure that the underlying implementation type is hidden and that distinct types are not identified during type checking. The stamp formalism meshes well with the dynamic semantics of Standard ML given in *The Definition*. Evaluation is defined on "type-erased" expressions; types have no significance during evaluation. Consequently, no management of stamps is required in the dynamic semantics.

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674. The second author was also partly supported by the US Army Research Office under Grant No. DAAH04-94-G-0289. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government or the National Science Foundation.

[1] See Cardelli's overview of type systems [3] for a survey and references to the literature.

However, recent work has emphasized *typed* dynamic semantics in which, for example, polymorphic values take type parameters and primitive operations may analyze types at run-time [26, 2]. The typed framework has several advantages. One is that it is strictly more general than the untyped framework since it is always possible to ignore type information on explicitly-typed terms, whereas we have no choice in the matter if we start with "type-erased" terms. Another is that the typed framework provides the link to type-based compilation methods mentioned earlier. These implementations rely critically on a type-passing dynamic semantics to achieve efficient data representation.

It is not immediately clear how to extend the stamps formalism of *The Definition* to an explicitly-typed setting. The difficulty is that primitive operations are defined on the underlying *representation* of an abstract type. For example, the fundamental operations on lists must know how lists are represented in order to distinguish between empty and non-empty lists and in order to retrieve the head and tail components in the latter case. Thus abstraction boundaries must be broken at run-time; consequently, an association between the abstract stamps and their underlying representations must be maintained.

In this paper we outline an interpretation of Standard ML in an explicitly-typed framework. The interpretation takes the form of a translation of Standard ML into an explicitly-typed λ-calculus. The target of the translation we call the *internal language*, or *IL*; the source language is then called the *external language*. The external language is the 1996 revision of Standard ML, as described in the revised *Definition* [17]. The internal language is derived from the XML language of Harper and Mitchell [9], but with a richer collection of primitive types and a more expressive module system based on the *translucent sum* [8], or *manifest type* [12], formalism. The internal language is given a type-passing dynamic semantics in the form of a transition system between states of an abstract machine.

The translation is presented by a set of inference rules reminiscent of the static semantics given in *The Definition*, with the internal language playing the role of the static semantic objects of *The Definition*. The translation rules typically define the translation of a phrase in terms of the translation of its constituent phrases, subject to context-sensitive constraints expressed by the internal language type system. Context-sensitive formation constraints are expressed by type checking constraints on the translation. Type propagation is controlled by a combination of the translucent sum formalism together with the representation of abstract types as modules with opaque type components. The internal language ensures that abstraction is respected, and, moreover, provides the requisite association of the abstract type to its representation required at run-time.

The interpretation described below has application both to semantics and to implementation. From the point of view of semantics, the internal language plays a role analogous to Scott's LAMBDA language for denotational semantics [24]. The meaning of a Standard ML program is obtained by translating the program into the internal language, which is given meaning by some other means. In our setting this is achieved by providing an operational semantics for the internal language, whereas in denotational semantics this is achieved by assigning a domain-theoretic interpretation to the target of the translation. Although we know of no complete domain-theoretic account of Standard ML, we conjecture that one could give such an interpretation by providing an adequate denotational semantics for the internal language.

The interpretation given here also has application to implementation. The TIL/ML compiler [26], and recent versions of the SML/NJ compiler [25], both take as their starting point a type-theoretic interpretation of Standard ML of the kind considered here. The TIL/ML compiler for Standard ML is based on the interpretation described here. The elaboration phase of the compiler is a direct "determinization" of the rules given below. Preliminary results [26] indicate that type-based translation is an effective method for reducing both the time and space requirements of Standard ML as compared to the type-free approach taken in earlier versions of the SML/NJ compiler.

Finally, we note that the internal language is intended to capture the fundamental constructs shared by many programming languages, much as Scott's LAMBDA is intended as a broadly-applicable meta-language for denotational semantics. To ensure maximal generality we have refrained from introducing constructs specific to the source language (in this case, Standard ML). In fact, we conjecture that languages such as Caml (a dialect of ML), Haskell (a lazy functional language), and Scheme (a dialect of Lisp) could be interpreted into an internal language substantially similar to the one we give here. This allows us to share meta-theoretic results (such as soundness of the internal language) among language definitions, and supports the construction of generic compilers for many different languages. To take one example, we may interpret

Judgment...	*Meaning...*
$\vdash decs$ ok	$decs$ is well-formed
$decs \vdash dec$ ok	dec is well-formed
$decs \vdash bnd : dec$	bnd has declaration dec
$decs \vdash knd : \mathsf{Kind}$	knd is well-formed
$decs \vdash con : knd$	con has kind knd
$decs \vdash con \equiv con' : knd$	constructor equivalence at kind knd
$decs \vdash exp : con$	exp has type con
$decs \vdash sdecs$ ok	$sdecs$ is well-formed
$decs \vdash sig : \mathsf{Sig}$	sig is well-formed
$decs \vdash sdecs \leq sdecs'$	component-wise subtyping
$decs \vdash sig \leq sig' : \mathsf{Sig}$	signature subtyping
$decs \vdash sdecs \equiv sdecs'$	component-wise equivalence
$decs \vdash sig \equiv sig' : \mathsf{Sig}$	signature equivalence
$decs \vdash sbnds : sdecs$	$sbnds$ has declaration list $sdecs$
$decs \vdash mod : sig$	mod has signature sig
$decs \vdash exp \downarrow con$	exp is valuable with type con
$decs \vdash mod \downarrow sig$	mod is valuable with signature sig

Figure 1: Judgments of the Internal Language Static Semantics

Scheme into the internal language by translating Scheme expressions into internal language expressions whose type is the recursive type corresponding to the Scheme value space. It is interesting to note that the interpretation framework is well-suited to languages such as Scheme that leave the order of evaluation of arguments unspecified — the translation can exploit the indeterminacy of the relational framework to "guess" an evaluation order. In contrast "direct" operational semantics for Scheme are unable to handle this aspect of the language.

2 The Internal Language

The internal language is an explicitly-typed λ-calculus with two levels, a *core* level and a *module* level. The two levels are linked by the ability to declare a module within a core-level expression. The internal language is based loosely on Harper and Mitchell's XML language [9], but with a treatment of modules derived from Harper and Lillibridge's [8, 14] translucent sum formalism and Leroy's manifest type system [12].

This section consists of a brief overview of the internal language. The language is defined by a set of inference rules for deriving the judgment forms given in Figure 1. A selection of the rules is given in Appendix B; the remainder can be found in a companion technical report [10]. For further background and motivation the reader is urged to consult the references cited above.

2.1 Constructors and Kinds

The syntax of *constructors* and *kinds* is given in Figure 2. Kinds classify constructors. Constructors of kind Ω are called *types*. Kinds are closed under formation of record kinds and function kinds.

$con ::=$	var	type variables	$rdecs ::=$	$\cdot$	empty
$\mid$	$\mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Char} \mid \cdots$	base types	$\mid$	$rdecs, rdec$	sequence
$\mid$	$\{rdecs\}$	record type	$rdec ::=$	$lab{:}con$	record field type
$\mid$	$con\ \mathsf{Ref}$	reference type			
$\mid$	$con \rightharpoonup con'$	partial function type	$knd ::=$	Ω	kind of types
$\mid$	$con \to con'$	total function type	$\mid$	$\{lab_1{:}knd_1, \ldots\}$	record kinds
$\mid$	Tagged	extensible sum type	$\mid$	$knd \Rightarrow knd'$	function kinds
$\mid$	$con\ \mathsf{Tag}$	exception-tag type			
$\mid$	$\Sigma_{\langle lab \rangle}\ (lab_1 \mapsto con_1, \ldots)$	(labelled) sum type			
$\mid$	$mod_{\mathrm{v}}.lab$	module projection			
$\mid$	$\lambda var{:}knd.con$	constructor function			
$\mid$	$\mu\ con$	constructor fixed-point			
$\mid$	$con\ con'$	constructor application			
$\mid$	$\{lab_1 {=} con_1, \ldots\}$	constructor records			

Figure 2: Constructors and Kinds

The internal language types include various base types, record types, sum types, partial and total function types, recursive types, reference types, tag types, and a type of tagged values. There is no subtyping at the core level, which is consistent with the lack of subtyping in the SML core language. Most of the type constructors are relatively standard, except for total function types, tag types, and the type of tagged values.

We borrow from the computational λ-calculus [19] an abstract notion of "definedness", called *valuability*, and the closely-associated notion of "totality" for functions. Valuability of core- and module-level expressions is expressed by a judgment form expressing that the given expression may be evaluated without engendering any computational effect. For decidability reasons, the rules define a conservative approximation of valuability. The approximation is strong enough to express the "value restriction" on polymorphism [29, 11] in Standard ML, as discussed in Section 3. In particular, all canonical forms are valuable, as are all variables (the dynamic semantics is call-by-value), and any application of a total function to a valuable argument. Total functions include primitives such as record field selection and those partial functions whose bodies are deemed valuable.

The internal language type Tagged is a type of dynamically-tagged values, corresponding to the external language type `exn`. The dynamically-generated tags are similar to the "names" considered by Pitts and Stark [21], except that we associate a type with each name to ensure type safety. Tags of values of type *con* are themselves values of type *con* Tag, indicating the type of the tagged value.

2.2 Expressions

The syntax of internal language expressions is given in Figure 3. Expressions are annotated with sufficient type information to ensure that each expression has a unique type.

Most of the expression forms are familiar from the λ-calculus literature. The treatment of functions is somewhat unusual, in order to account for mutually-recursive functions in a call-by-value setting. An expression of the form

$$\begin{array}{ll} \mathsf{fix} & var'_1(var_1{:}con_1){:}con'_1 \mapsto exp_1, \\ & \qquad\vdots \\ & var'_n(var_n{:}con_n){:}con'_n \mapsto exp_n \\ \mathsf{end} & \end{array}$$

represents a "tuple" of n mutually-recursive functions. This expression, as well as any projection from it, is valuable.

New exception tags of type *con* Tag are created using the expression form new_tag. An expression injected into the type Tagged with a particular tag by the tag form. The corresponding "projection" iftagof checks for a specified tag and if found extracts the underlying value.

$exp ::=$	$scon$	constants
$\mid$	var	variables
$\mid$	loc	memory locations
$\mid$	tag	exception tags
$\mid$	$\mathsf{fix}\ fbnds\ \mathsf{end}$	recursive functions
$\mid$	$exp\ exp'$	application
$\mid$	$\{rbnds\}$	record expression
$\mid$	$\pi_{lab}\ exp$	record projection
$\mid$	$\mathsf{handle}\ exp\ \mathsf{with}\ exp'$	handle exception
$\mid$	$\mathsf{raise}^{con}\ exp$	raise exception
$\mid$	$\mathsf{ref}^{con}\ exp$	new ref cell
$\mid$	$\mathsf{get}\ exp$	dereference
$\mid$	$\mathsf{set}\ (exp, exp')$	assignment
$\mid$	$\mathsf{roll}^{con}\ exp$	coerce to μ type
$\mid$	$\mathsf{unroll}^{con}\ exp$	coerce from μ type
$\mid$	$\partial\ exp$	coerce partial to total

$\mid$	$\mathsf{inj}_{lab}^{con}\ exp$	injection into sum
$\mid$	$\mathsf{proj}_{lab}^{con}\ exp$	sum projection
$\mid$	$\mathsf{case}^{con}\ exp\ \mathsf{of}\ exp_1, \ldots\ \mathsf{end}$	sum case analysis
$\mid$	$\mathsf{new\_tag}[con]$	extend type Tagged
$\mid$	$\mathsf{tag}(exp, exp)$	injection into Tagged
$\mid$	$\mathsf{iftagof}\ exp\ \mathsf{is}\ exp'$ $\mathsf{then}\ exp''\ \mathsf{else}\ exp'''$	tag analysis
$\mid$	$exp_1 =_{\mathsf{Int}} exp_2, \ldots$	base equalities
$\mid$	$mod.lab$	module projection
$rbnds ::=$	$\cdot$	empty
$\mid$	$rbnds, rbnd$	sequence
$rbnd ::=$	$lab{=}exp$	record field binding
$fbnds ::=$	$\cdot$	empty
$\mid$	$fbnds, fbnd$	sequence
$fbnd ::=$	$var'(var{:}con){:}con' \mapsto exp$	function binding
$labs ::=$	$lab \mid labs.lab$	sequence of labels
$path ::=$	$var \mid var.labs$	qualified variable

Figure 3: Expressions

Reference types are built into the internal language to avoid unnatural encodings. The operations ref, get, and set of the internal language correspond directly to the operations `ref`, `!`, and `:=`, respectively, of SML.

For similar reasons an exception mechanism is built into the internal language. Exceptions carry values of a specific type, which is taken here to be the type Tagged to be consistent with Standard ML, but we note that there is no essential connection between the type Tagged and the exception mechanism *per se*. We could as well consider exception values of any fixed type, or even have several different exception mechanisms, each carrying values of a type specific to that form of exception.

The core and module levels of the language are linked by the expression form for module component selection *mod.lab*. Allowing *mod* to be an arbitrary module means that "let-polymorphism" is definable in our internal language. More importantly, the bindings that may occur in a such let are exactly those that may occur in a structure, and we have the ability to define modules *within* an expression. This is exploited heavily in the interpretation of Standard ML given in Section 3.

2.3 Modules and Signatures

The module language is based on the *translucent sum* (or *manifest type*) formalism [8, 12]. The syntax for modules and signatures is given in Figure 4. The basic form of module is a *structure*, which consists of a sequence of constructor, expression, and module bindings. *Structure signatures* consist of a corresponding sequence of constructor, expression, and module declarations. The module system is closed under formation of *functors*, which are functions mapping modules to modules. *Functor signatures* are dependent function types describing the result of a functor in terms of its argument. Modules are "second-class" — there are no conditional module expressions, nor may modules be stored in reference cells or returned from core-level functions.

The main characteristic of the internal language module calculus is the reliance on signatures to mediate inter-module dependencies — the formation of a module expression relies only on the interface, and not the implementation, of any modules on which it depends. Propagation of type sharing information is managed by the selective exposure of type information in a signature through the use of *transparent* and *opaque* type specifications. Translucent sums may be seen as a generalized form of existential type [18] that affords

mod ::=	var	module variable
|	$[sbnds]$	structure
|	$\lambda var{:}sig.mod$	functor
|	$mod\ mod'$	functor application
|	$mod.lab$	structure projection
|	$mod{:}sig$	signature ascription
$sbnds$ ::=	$\cdot$	structure field bindings
|	$sbnds, sbnd$	
$sbnd$ ::=	$lab{:}bnd$	
bnd ::=	$var{=}con$	constructor binding
|	$var{=}exp$	expression binding
|	$var{=}mod$	module binding

sig ::=	$[sdecs]$	structure signature
|	$(var{:}sig) \rightharpoonup sig'$	partial functor signature
|	$(var{:}sig) \rightarrow sig'$	total functor signature
$sdecs$ ::=	$\cdot$	structure field dec.
|	$sdecs, sdec$	
$sdec$ ::=	$lab{:}dec$	
$decs$ ::=	$\cdot$	declaration lists
|	$decs, dec$	
dec ::=	$var{:}con$	expression variable dec.
|	$var{:}sig$	module variable dec.
|	$var{:}knd$	opaque type dec.
|	$var{:}knd{=}con$	transparent type dec.
|	$loc{:}con$	typed locations
|	$tag{:}con$	typed exception tag

Figure 4: Modules and Signatures

fine-grained control over the "degree" of abstractness of a type. They may also be seen as a variant of the "dependent sum" type [15], adopting the flexible "projection" notation for component selection, but avoiding implementation dependencies.

Structure signatures consist of a sequence of constructor, value, and module declarations. Constructor declarations may either be opaque (specifying only a kind) or transparent (specifying the identity of the constructor). Value declarations specify the type of a value component, and module declarations specify the signature of a module component. Each declaration specifies an *internal name* and an *external name* for that component. The internal name is used to express dependencies of one declaration on another. For example, the type of a value component may refer (via the internal name) to a type declared earlier in the signature, or the definition of a constructor may refer to previously-declared constructors. Internal names are bound variables introduced at the point of declaration; they may be freely renamed within their scope without changing the meaning of the signature. The external name of a component is a label; structure components are accessed using these labels. External names are not variables; they may not be renamed without changing the meaning of the signature.

For example, the signature $[T{\triangleright}t{:}\Omega, U{\triangleright}u{:}\Omega{=}t{\times}t, X{\triangleright}x{:}u]$, describes a module with two type components, with external names T and U, and internal names t and u, respectively, and one value component, with external name X and internal name x. The type component U is defined to be equal to the product of the T component with itself, and the X component has type U. Notice that the dependencies are expressed using the internal names.

Every module value possesses a most-specific signature in which the identity of all type components is propagated using transparent type bindings. For example, the most specific signature for the structure

$$[T{\triangleright}t{=}\mathsf{Int}, U{\triangleright}u{=}\mathsf{Int}{\times}\mathsf{Int}, X{\triangleright}x{=}(3,4)]$$

is given by

$$[T{\triangleright}t{:}\Omega{=}\mathsf{Int}, U{\triangleright}u{:}\Omega{=}\mathsf{Int}{\times}\mathsf{Int}, X{\triangleright}x{:}\mathsf{Int}\times\mathsf{Int}].$$

Modules may be given less-specific signatures using *subsumption* — the signature of a module may be weakened to a "larger" signature in the sub-signature ordering. This ordering is a non-coercive, forgetful ordering in which signatures may be weakened by neglecting type definitions, rendering opaque one or more transparent components. For example, using subsumption we may assign the less informative signatures $[T{\triangleright}t{:}\Omega, U{\triangleright}u{:}\Omega{=}t{\times}t, X{\triangleright}x{:}u]$ and $[T{\triangleright}t{:}\Omega{=}\mathsf{Int}, U{\triangleright}u{:}\Omega, X{\triangleright}x{:}u]$ to the module expression given above.

A module may be "sealed" by signature *ascription*. The module expression $mod{:}sig$ is well-formed if mod has the signature sig (possibly through a use of subsumption). Then $mod{:}sig$ has most-specific signature sig. In practice we use ascription to make type components of a module abstract.

Parameterized modules, or *functors*, are written using the familiar λ-notation; there are no recursive functors. *Functor signatures* are a form of "Π type" (dependent function type) in which the signature of the result depends on the argument to the functor. This is used to express the propagation of type sharing properties from the argument to the result, without relying on exposure of the implementation of the functor. The sub-signature relation is extended to functor signatures in the usual way, contravariantly in the domain and covariantly in the codomain [2]. Only non-dependent functors may be applied to arguments; the dependency must first be eliminated through the use of the sub-signature and signature equivalence relations. This is always possible if the argument is valuable, but may not be possible in general.[2]

For example, suppose we have a functor

$$f : (u{:}[T \triangleright t{:}\Omega, X \triangleright x{:}t]) \rightharpoonup [T' \triangleright t'{:}\Omega{=}u.T \times u.T, X' \triangleright x'{:}t']$$

and we wish to apply this functor to a structure with signature $[T \triangleright t{:}\Omega{=}\mathsf{Float}, X \triangleright x{:}\mathsf{Float}]$. This is possible because by subsumption `f` also satisfies the non-dependent signature:

$$u{:}[T \triangleright t{:}\Omega{=}\mathsf{Float}, X \triangleright x{:}\mathsf{Float}] \rightharpoonup [T' \triangleright t'{:}\Omega{=}\mathsf{Float} \times \mathsf{Float}, X' \triangleright x'{:}\mathsf{Float} \times \mathsf{Float}]$$

showing that the application will have signature $[T' \triangleright t'{:}\Omega{=}\mathsf{Float} \times \mathsf{Float}, X' \triangleright x'{:}\mathsf{Float} \times \mathsf{Float}]$.

As in the core language, module expressions are categorized as valuable or non-valuable. Functors whose bodies are valuable module expressions are said to be *total*; all others are partial. Modules whose components are all valuable are themselves valuable, as are all module variables, and all selections of module components from valuable modules. An *ascription* of a signature to a module, written $mod{:}sig$, is valuable if the underlying module is valuable, but is not a value. Since type components may only be selected from module *values*, this convention ensures that abstraction boundaries are respected. If a signature is ascribed to a module, then its abstract type components may only be accessed by first binding that module to a variable, then selecting from that variable. This ensures that the abstraction boundary imposed by the ascription is respected, and ascribing the same signature to the same module will yield incompatible abstract types.

2.4 Dynamic Semantics

The dynamic semantics of the internal language is a call-by-value operational semantics presented as a rewriting relation between states of an abstract machine. The presentation is strongly influenced by the work of Plotkin [22] and Wright and Felleisen [30], but is departure from the framework of *The Definition of Standard ML*. The state-machine presentation avoids the need for implicit evaluation rules for handling exceptions, and supports a natural interpretation of type soundness that does not rely on artificial "wrong" transitions. We prefer to use substitution, rather than environments, because this allows us to regard values as particular forms of expression; this also simplifies the statement of soundness, particularly in the presence of references. We maintain a store for assignable cells and dynamically-generated tags, as in *The Definition*, but, in addition, we maintain an explicit store and tag typing context, in keeping with our explicitly-typed framework.

Each state Σ of the abstract machine is a triple of the form (Δ, σ, exp), where

- Δ is a typing context ($decs$) for locations and tags created at run-time. This maintains a record of what exception tags and locations have already been allocated, and is also used in our soundness proofs.
- σ is a finite mapping from locations typed in Δ to expression values (exp_{V}). The syntax of all values appears in Figure 5.
- exp is an expression.

[2] For the purposes of the interpretation of Standard ML, we will ensure that functor dependencies can always be eliminated through simple uses of signature equivalence.

$$
\begin{array}{rcl}
exp_{\mathsf{v}} & ::= & scon \\
& | & loc \\
& | & tag \\
& | & path \\
& | & \{rbnds_{\mathsf{v}}\} \\
& | & \mathsf{fix}\, fbnds\, \mathsf{end} \\
& | & \pi_{\overline{k}}\, \mathsf{fix}\, fbnds\, \mathsf{end} \\
& | & \mathsf{inj}_i^{con}\ exp_{\mathsf{v}} \\
& | & \mathsf{tag}(tag, exp_{\mathsf{v}}) \\
& | & \mathsf{roll}^{con}\ exp_{\mathsf{v}} \\
& | & \partial\ exp_{\mathsf{v}} \\
& | & \mathsf{forget\_label}\ exp_{\mathsf{v}} \\
rbnds_{\mathsf{v}} & ::= & \cdot \\
& | & rbnds_{\mathsf{v}}, rbnd_{\mathsf{v}} \\
rbnd_{\mathsf{v}} & ::= & lab{=}exp_{\mathsf{v}} \\
\\
val & ::= & exp_{\mathsf{v}} \\
& | & mod_{\mathsf{v}} \\
& | & con
\end{array}
\qquad
\begin{array}{rcl}
mod_{\mathsf{v}} & ::= & path \\
& | & [sbnds_{\mathsf{v}}] \\
& | & \lambda var{:}sig.mod \\
\\
bnd_{\mathsf{v}} & ::= & var{=}exp_{\mathsf{v}} \\
& | & var{=}mod_{\mathsf{v}} \\
& | & var{=}con \\
\\
sbnds_{\mathsf{v}} & ::= & \cdot \\
& | & sbnds_{\mathsf{v}}, sbnd_{\mathsf{v}} \\
\\
sbnd_{\mathsf{v}} & ::= & lab{:}bnd_{\mathsf{v}}
\end{array}
$$

Figure 5: Internal Language Values

A state is *terminal* if it has one of the following forms:

$$
\begin{array}{ll}
(\Delta, \sigma, exp_{\mathsf{v}}) & \text{normal termination} \\
(\Delta, \sigma, \mathsf{raise}^{con}\ exp_{\mathsf{v}}) & \text{uncaught exception}
\end{array}
$$

All other states are *nonterminal*. We let Σ_t range over terminal states.

The dynamic semantics is a transition relation $\Sigma \hookrightarrow \Sigma'$ between states, defined by the rules given in Appendix C. As usual, we denote the reflexive, transitive closure of $\hookrightarrow$ by $\hookrightarrow^*$. The rules defining the relation have the form

$$(\Delta, \sigma, exp) \hookrightarrow (\Delta', \sigma', exp'),$$

possibly with some side conditions. The rules make use of the notion of an "evaluation context", an expression or module with a single "hole" (see Figure 6). The expression $E[phrase]$ is the expression resulting from replacing the hole in E by *phrase*. We use R to denote an expression context constructed from the grammar in Figure 6 without the form $\mathsf{handle}\ E\ \mathsf{with}\ exp$.

Most of the rules of the dynamic semantics are straightforward interpretations of the constructs of the internal language. Exceptions are handled using explicit "jumps" though evaluation contexts that do not involve exception handlers. This is achieved by relying on a form of pattern-matching to capture the informal idea of jumping to the nearest enclosing exception handler. Tags and reference cells are explicitly allocated during evaluation, and their types are maintained in the state. Uses of the sub-signature relation have no run-time significance; control over type sharing properties is entirely a matter of static checking.

As a technical convenience, for the purpose of the dynamic semantics we include a CHAM-like structural equivalence rule for structures, extending the standard equivalence of terms or modules up to alpha-conversion. This is generated by the schema

$$[sbnds, lab\triangleright var{=}val, sbnds'] \equiv [sbnds, lab\triangleright var{=}val, \{val/var\}sbnds']$$

which allows us to remove dependencies between fields in a structure when the dependency is on a field carrying a value. Factoring out such substitutions separately simplifies the dynamic semantics, but is not critical to the framework.

$$
\begin{array}{rll}
E ::= & E\ exp & \mid\ \mathsf{tag}(E, exp) \\
\mid & exp_{\mathsf{v}}\ E & \mid\ \mathsf{tag}(exp_{\mathsf{v}}, E) \\
\mid & \{rbnds_{\mathsf{v}}, lab{=}E, rbnds\} & \mid\ \mathsf{iftagof}\ E\ \mathsf{is}\ exp\ \mathsf{then}\ exp'\ \mathsf{else}\ exp'' \\
\mid & \pi_{lab}\ E & \mid\ \mathsf{iftagof}\ exp_{\mathsf{v}}\ \mathsf{is}\ E\ \mathsf{then}\ exp'\ \mathsf{else}\ exp'' \\
\mid & \mathsf{handle}\ E\ \mathsf{with}\ exp & \mid\ E{=}_{con}\, exp \\
\mid & \mathsf{raise}^{con}\ E & \mid\ exp_{\mathsf{v}}{=}_{con}\, E \\
\mid & \mathsf{ref}^{con}\ E & \mid\ [sbnds_{\mathsf{v}}, lab{\triangleright}var{=}E, sbnds] \\
\mid & \mathsf{get}\ E & \mid\ E\ mod \\
\mid & \mathsf{set}\ (E, exp) & \mid\ mod_{\mathsf{v}}\ E \\
\mid & \mathsf{set}\ (exp_{\mathsf{v}}, E) & \mid\ E.lab \\
\mid & \mathsf{roll}^{con}\ E & \mid\ E{:}sig \\
\mid & \mathsf{unroll}^{con}\ E & \\
\mid & \mathsf{inj}_i^{con}\ E & \\
\mid & \mathsf{proj}_i^{con}\ E & \\
\mid & \mathsf{case}^{con}\ E\ \mathsf{of}\ exp_1, \ldots, exp_n\ \mathsf{end} &
\end{array}
$$

Figure 6: Evaluation Contexts

2.5 Properties of the Internal Language

In order to relate the static and dynamic semantics, we must first state some technical properties of the operational semantics.

We define two states to be *equivalent*, written

$$(\Delta, \sigma, exp) \cong (\Delta', \sigma', exp'),$$

if they are component-wise equal up to consistent renaming of the locations and exception tags appearing in Δ and of bound variables in the expression component.

Proposition 1 (Determinacy of Evaluation)
The following properties hold:

1. *If Σ is terminal and $\Sigma \cong \Sigma'$, then Σ' is also terminal.*
2. *If $\Sigma \hookrightarrow \Sigma_1$ and $\Sigma \cong \Sigma'$, then there exists a state $\Sigma_1' \cong \Sigma_1$ such that $\Sigma' \hookrightarrow \Sigma_1'$.*
3. *If $\Sigma_1 \hookrightarrow \Sigma_1'$, $\Sigma_2 \hookrightarrow \Sigma_2'$ and $\Sigma_1 \cong \Sigma_2$ then $\Sigma_1' \cong \Sigma_2'$.*
4. *If $\Sigma \hookrightarrow^* \Sigma_t$ and $\Sigma \hookrightarrow^* \Sigma_t'$ then $\Sigma_t \cong \Sigma_t'$.*

The following proposition states that internal language judgments are preserved under substitution of values for free variables in a typing judgment, where a value is defined syntactically in Figure 5 to be a phrase in evaluated form.

Proposition 2 (Decomposition & Replacement)

1. *If $decs \vdash E[exp] : con$ and exp is closed, then $decs \vdash exp : con'$ for some type con'. Furthermore, if $decs \vdash exp' : con'$ where exp' is closed, then $decs \vdash E[exp'] : con$.*
2. *If $decs \vdash E[mod] : con$ and mod is closed, then $decs \vdash mod : sig$ for some signature sig. Furthermore, if $decs \vdash mod' : sig$ where mod' is closed, then $decs \vdash E[mod'] : con$.*

Following Harper [7] and Wright and Felleisen [30], we say that a store σ is well-formed with respect to a context Δ, written $\Delta \vdash \sigma$, if

$$\forall loc \in \mathrm{BV}(\Delta),\ \text{if } \Delta \vdash loc : con\ \mathsf{Ref}\ \text{then } \Delta \vdash \sigma(x) : con.$$

This formulation of store typing avoids the need for complex maximal fixed point constructions [27].

Fix a base type *ans* of *answers* to which a complete, closed program might evaluate.[3] We can say that a machine state is well-formed, written

$$\vdash (\Delta, \sigma, exp),$$

if and only if $\Delta \vdash exp : ans$, exp has no free (expression, constructor, or module) variables, and $\Delta \vdash \sigma$.

Well-formedness of a state is preserved by evaluation.

Proposition 3 (Preservation)
If $\vdash (\Delta, \sigma, exp)$ *and* $(\Delta, \sigma, exp) \hookrightarrow (\Delta', \sigma', exp')$ *then* $\vdash (\Delta', \sigma', exp')$.

Evaluation can never "get stuck": if a well-formed state is not terminal, then there is always an applicable transition to another (well-formed) state. The proof relies on a characterization of the shapes of closed values of each type.

Proposition 4 (Canonical Forms)

1. *Assume* $\Delta \vdash {exp_{\mathrm{V}}}' : con'$ *where* ${exp_{\mathrm{V}}}'$ *is closed.*

If con' *is of the form...*	*then* ${exp_{\mathrm{V}}}'$ *is of the form...*
$con_1 \rightharpoonup con_2$	$\pi_{\overline{k}}\,\mathsf{fix}\ fbnds\ \mathsf{end}$
$con_1 \rightarrow con_2$	$\pi_{\overline{1}}\,\mathsf{fix}\ fbnd\ \mathsf{end}$
$\{lab_1{:}con_1, \cdots, lab_n{:}con_n\}$	$\begin{cases} \{lab_1{=}exp_{\mathrm{V}\,1}, \cdots, lab_n{=}exp_{\mathrm{V}\,n}\} \\ \mathsf{fix}\ fbnds\ \mathsf{end} \end{cases}$
$\Sigma_{\langle lab_i \rangle}\,(lab_1{\mapsto}con_1, \ldots, lab_n{\mapsto}con_n)$	$\mathsf{inj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\ exp_{\mathrm{V}}$
$(\pi_i\ (\mu\ con))\ con'$	$\mathsf{roll}^{(\pi_i\ (\mu\ con))\ con'}\ exp_{\mathrm{V}}$
Tagged	$\mathsf{tag}(tag, exp_{\mathrm{V}})$
$con\ \mathsf{Ref}$	loc
base type	$scon$

2. *Assume* $\Delta \vdash {mod_{\mathrm{V}}}' : sig'$ *where* ${mod_{\mathrm{V}}}'$ *is closed.*

If sig' *is of the form...*	*then* ${mod_{\mathrm{V}}}'$ *is of the form...*
$[sdecs]$	$[sbnds_{\mathrm{V}}]$
$var{:}sig \rightharpoonup sig'$	$\lambda var{:}sig.mod$
$var{:}sig \rightarrow sig'$	$\lambda var{:}sig.mod$

Proposition 5 (Progress)
Let $\Sigma = (\Delta, \sigma, exp)$. *If* $\vdash \Sigma$ *then either* Σ *is terminal or there exists a state* Σ' *such that* $\Sigma \hookrightarrow \Sigma'$.

3 Elaboration of Standard ML into the Internal Language

The type-theoretic interpretation of Standard ML takes the form of a set of inference rules for deriving *elaboration judgments* of the form

$$\Gamma \vdash \textit{EL-phrase} \leadsto phrase : class.$$

Here *EL-phrase* is a phrase of the Standard ML abstract syntax, *phrase* is its translation into the internal language, and *class* is an internal-language kind, type, or signature classifying *phrase*. The context Γ associates external names and classifiers to internal names. A complete list of the judgment forms constituting the interpretation are given in Figure 7.

The elaboration of Standard ML into the internal language involves the following major steps:

[3] A reasonable choice might be String, or Unit if we model all I/O by updating the store; the particular choice does not affect our results.

Judgment...	*Meaning...*
$\Gamma \vdash expr \leadsto exp : con$	expression
$\Gamma \vdash match \leadsto exp : con$	pattern match
$\Gamma \vdash strdec \leadsto sbnds : sdecs$	declaration
$\Gamma \vdash strexp \leadsto mod : sig$	structure expression
$\Gamma \vdash spec \leadsto sdecs$	signature specification
$\Gamma \vdash sigexp \leadsto sig : \mathsf{Sig}$	signature expression
$\Gamma \vdash ty \leadsto con : \Omega$	type expression
$\Gamma \vdash tybind \leadsto sbnds : sdecs$	`type` definition
$\Gamma \vdash datbind \leadsto sbnds : sdecs$	`datatype` definition
$\Gamma \vdash_{\overline{\mathrm{ctx}}} labs \leadsto path : class$	lookup in Γ
$decs; path{:}sig \vdash_{\overline{\mathrm{sig}}} labs \leadsto labs' : class$	lookup in signature
$decs \vdash_{\overline{\mathrm{inst}}} \leadsto [sbnds_{\mathrm{v}}] : [sdecs']$	polymorphic instantiation
$\Gamma \vdash pat \Leftarrow exp : con$ else $exp' \leadsto sbnds : sbnds$	pattern compilation
$decs \vdash_{\overline{\mathrm{eq}}} con \leadsto exp_{\mathrm{v}}$	equality compilation
$decs \vdash_{\overline{\mathrm{sub}}} path : sig_0 \preceq sig \leadsto mod : sig'$	coercion compilation
$sig \vdash_{\overline{\mathrm{wt}}} labs := con : knd \leadsto sig' : \mathsf{Sig}$	impose definition
$sig \vdash_{\overline{\mathrm{sh}}} labs := labs' : knd \leadsto sig' : \mathsf{Sig}$	impose sharing

Figure 7: Judgment forms for the Elaboration

1. **Identifier resolution**. External-language identifiers are translated into internal-language paths according to the scoping rules of Standard ML. Re-defined identifiers are renamed to avoid conflicts.

2. **Type checking and type reconstruction**. The elaboration rules ensure that the translation of an external-language phrase is well-formed with a specified classifier (kind, type, or signature). Implicit type information — such as type labels on variables and polymorphic abstraction and instantiation — is made explicit. Polymorphic abstractions are represented as internal-language functors.

3. **Datatype and pattern-matching translation**. Datatype declarations are translated into modules with an opaque implementation type and operations for creating and destructuring values of this type. Patterns are compiled into uses of these operations, along with record projections and equality tests.

4. **Equality compilation**. For types that admit equality, a canonical equality operation is generated and passed as required. Equality polymorphic operations are represented as functors taking both the type together with the associated equality operation. Datatypes that admit equality are equipped with an equality operation.

5. **Signature matching**. The *instantiation* ordering — arising from the presence or absence of type definitions in signatures — is managed by the sub-signature relation of the internal language. The *enrichment* ordering — arising from the ability in Standard ML to drop or re-order module components — is handled by an explicit coercion operation generated by the elaborator. Since we are working with an explicitly-typed internal language, polymorphic instantiation in signature matching is also managed by explicit coercion.

6. **Sharing expansion**. Uses of type sharing specifications are expanded into uses of type definitions in signatures [13]. The `where type` construct of Standard ML is translated by explicitly "patching" internal-language signatures.

7. **Generativity and persistence**. In Standard ML type identifiers may persist beyond their apparent scope of definition. This is managed here by the restriction to "named form" programs at the module level (according to which all modules must be bound to identifiers before use), and an explicit mechanism for retaining types through renaming when they appear to go out of scope.

The elaboration process is discussed in more detail in the remainder of this section.

3.1 Identifier Resolution

A fundamental task of elaboration is associating internal-language paths to external-language identifiers. Since the external language permits shadowing of identifiers, we cannot assume a fixed correspondence between Standard ML identifiers and internal-language variables. Therefore we translate identifiers into internal-language paths, and the correspondence is maintained by an elaboration context. This context is essentially a sequence of internal-language structure field bindings, but with the possibility of duplicated labels due to shadowing. We may regard elaboration contexts as declaration lists by dropping the labels from the components (turning each *sdec* into its underlying *dec*); in this way the formation rules of the internal language determine validity of elaboration contexts.

We postulate an injection $\overline{\cdot}$ of ML identifiers into internal-language labels. The range of this mapping is assumed co-infinite in the set of labels, ensuring that we may choose arbitrarily many new labels not in the range of this mapping. We further assume that the labels "eq", "expose", "it," and "tag" are outside of the range of this mapping, and that identically-named identifiers from different external language namespaces (expression identifiers, type identifiers, signature identifiers, structure identifiers, *etc.*) are mapped to distinct labels. On the other hand, we assume a single namespace for external-language variables, datatype constructors, and exception constructors; in Standard ML these distinctions are not syntactically apparent and making this distinction falls to the elaboration itself.

The translation of an (possibly overbarred) long identifier into an path is expressed by the judgment

$$\Gamma \vdash_{\overline{\mathrm{ctx}}} \mathit{labs} \rightsquigarrow \mathit{path} : \mathit{class},$$

which looks up the sequence of labels *labs* in the elaboration context Γ and returns a path *path* with classifier *class*. The lookup rules describe a sequential search[4] from right-to-left, subject to a simple convention, called the *star convention*, for handling "open" structures. Labels marked with an asterisk are treated as names of open structures, whose bindings are implicitly available for use in a Standard ML phrase. A second set of lookup rules expresses identifier search within a structure; these rules are also used in translating Standard ML "long identifiers."

As a very simple example,

$$X \triangleright x{:}\mathsf{Int}, L \triangleright l{:}[X{:}\mathsf{Char}] \vdash_{\overline{\mathrm{ctx}}} X \rightsquigarrow x : \mathsf{Int},$$

but

$$X \triangleright x{:}\mathsf{Int}, L^{*} \triangleright l{:}[X{:}\mathsf{Char}] \vdash_{\overline{\mathrm{ctx}}} X \rightsquigarrow l.X : \mathsf{Char},$$

because the structure L is open and contexts (and signatures) are searched depth-first from right to left.

3.2 Expressions and Declarations

The general form of elaboration judgment for expressions is

$$\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}$$

where *expr* is an external-language expression, and *exp* is the corresponding internal-language expression having type *con*. These elaboration rules are shown in full in Appendix D.2. Identifiers are translated using the lookup rules mentioned above, and if found to be polymorphic are immediately instantiated (polymorphism is discussed in more detail below). Constructors (functions with total types) are translated to user-level functions (with partial types) when used as values. Application translates to internal-language

[4] In our compiler implementation, a more efficient algorithm is used.

application, with a check to make sure the translated application is well-formed. Record expressions are translated to internal-language records. Since Standard ML identifies record types under permutation of fields, the translation reorders these fields into a canonical ordering while preserving evaluation order. Explicit type constraints are verified, but do not appear in the translation. The exception expressions `raise` and `handle` translate into their internal-language equivalents. Function abstractions in the EL translate to function abstractions in the IL, wrapped to raise the `Match` exception in the case of pattern match failure. Equality comparisons invoke the equality compiler (also described below) to generate the appropriate equality operation.

An important invariant of the translation is that "syntactic values" in the Standard ML sense are translated to valuable expressions. This is necessary to enforce the value restriction on polymorphism, according to which only syntactic values may be polymorphically generalized. However, our treatment of pattern matching leads to a minor discrepancy between the interpretation given here and *The Definition of Standard ML*, as discussed in Section 3.5 below.

The general form of elaboration judgment for declarations is

$$\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}.$$

A selection of such judgments is shown in Appendix D.5. Each external-language declaration is translated into structure field bindings. For simple bindings, there is exactly one field binding for each identifier bound. In more complex cases — such as complex patterns or `datatype` declarations — the result will contain not only bindings for the identifiers explicitly involved in the declaration, but also "internal" fields used by the elaborator itself.

The `open` declaration (Rule 108) is regarded as the declaration of an "anonymous" substructure that is implicitly opened for identifier lookup using the "star convention" discussed above. In implementation terms this means that an `open` declaration requires only constant time and space, rather than time and space proportional to the size of the opened structure. To account for shadowing, declaration sequencing goes beyond simple concatenation of bindings by renaming fields corresponding to shadowed identifiers.

3.3 Polymorphism

Polymorphism is interpreted by explicit type abstraction and type application [9]. However, we do not treat type abstraction and application as a primitive notion (as in the polymorphic λ-calculus [4, 23]). Instead, we represent a polymorphic value as an internal-language functor abstracted on a structure whose components are types, yielding a structure with a single component labeled "it" for the value itself. This representation is consistent with the "second class" nature of both polymorphic values and modules in Standard ML. Moreover, it is especially natural in the presence of equality type variables, which we regard as structures consisting of a type and the corresponding equality operation (see Section 3.6 for further details).

For example, the polymorphic identity function is translated[5] to the functor

$$\lambda(s{:}[T{:}\Omega]).[\mathrm{it}{=}\lambda(x{:}s.T).x]$$

with signature

$$(s{:}[T{:}\Omega]){\to}[\mathrm{it}{:}s.T{\rightharpoonup}s.T].$$

Note that the functor is given a *total* functor type, expressing the fact that type instantiation does not engender an effect. This is consistent with the "value restriction" on polymorphism in Standard ML, according to which only syntactic values may be polymorphically abstracted.

3.4 Datatype Declarations

The treatment of datatypes is technically complex, but conceptually straightforward. A `datatype` declaration elaborates to a structure consisting of a type together with operations to create and analyze values of the type. If the datatype admits equality, then the structure contains an equality function as well. The

[5] We have simplified the translation slightly for the sake of readability.

$$\begin{array}{l}[\overline{\texttt{list}}\triangleright list{:}\Omega\Rightarrow\Omega,\\ \quad \text{eq}{:}(s{:}[T^*{:}[T\triangleright t{:}\Omega, \text{eq}{:}t\times t\rightharpoonup\mathsf{Bool}]])\to[\text{it}{:}list\,(s.T^*.T)\times list\,(s.T^*.T)\rightharpoonup\mathsf{Bool}]\\ \quad \overline{\texttt{Nil}}{:}(s{:}[T{:}\Omega])\to[\text{it}{:}list\,(s.T)],\\ \quad \overline{\texttt{Cons}}{:}(s{:}[T{:}\Omega])\to[\text{it}{:}s.T\times list\,(s.T)\to list\,(s.T)],\\ \quad \text{expose}{:}(s{:}[T{:}\Omega])\to[\text{it}{:}list\,(s.T)\to\Sigma(\mathsf{Unit}, s.T\times list\,(s.T))]]\end{array}$$

Figure 8: The signature $sig_{\texttt{list}}$

underlying implementation type is defined to be a recursive sum type, with one summand corresponding to each constructor in the declaration. The constructors are represented by total functions that inject values into the appropriate summand of the recursive type. The analysis operation exposes values of the abstract type as values of a corresponding sum type. The structure is "sealed" with a signature derived from the **datatype** declaration in which the implementation type is held abstract, and the operations described above are declared as operations on that abstract type. Holding the implementation type abstract captures the "generativity" of **datatype** declarations in Standard ML; the declared type is "new" in the sense that it is represented by a bound type variable that is, by α-conversion, distinct from all other types in the program. Analogously, **datatype** specifications (which may occur in signatures) are elaborated into the same signature used to seal the structure resulting from elaboration of the corresponding **datatype** declaration.

The treatment of datatypes is best illustrated by example. Viewed as a specification, the Standard ML phrase

```
datatype 'a list = Nil | Cons of 'a * 'a list.
```

elaborates to the signature $sig_{\texttt{list}}$ given in Figure 8. Viewed as a declaration, this phrase elaborates into the sealed structure $mod_{\texttt{list}}{:}sig_{\texttt{list}}$, where $mod_{\texttt{list}}$ is given in Figure 9.

The signature $sig_{\texttt{list}}$ describes a structure with five components, one corresponding to the `list` type constructor itself, one for `list` equality, two for the constructors, and one for deconstructing values of this type. The `list` type constructor is represented by a type operator of kind $\Omega\Rightarrow\Omega$. The operation eq is the equality function on lists; it takes a type T and an equality for values of type T, and returns the equality function for values of type $\overline{\texttt{list}}\,T$. The value constructor $\overline{\texttt{Nil}}$ is the polymorphic total function that, when given a type, creates the empty list of that type. Similarly $\overline{\texttt{Cons}}$ is the polymorphic total function to add an element to the front of a list. The polymorphic function expose exposes the underlying implementation of the datatype as a sum type for the purposes of destructuring.

The structure $mod_{\texttt{list}}$ implements the signature $sig_{\texttt{list}}$. The implementation is relatively straightforward, following the informal discussion above. We have elided the definition of equality on lists, but it corresponds directly to the obvious recursive definition which can be generated mechanically.

The account of datatypes given here differs from that in Standard ML in that we do not admit equality on "non-uniform" datatypes, for which we would need polymorphic recursion, which is not admitted in Standard ML or in our internal language. For example, the following declaration is legal in Standard ML, and the declared type admits equality:

```
datatype 'a t = A of 'a | B of ('a * 'a) t
```

Although this is an admissible declaration according to our elaboration rules, it does not admit equality in the translation due to the absence of polymorphic recursion (which would correspond to a recursive functor in our setting).

3.5 Pattern Compilation

Pattern compilation is the process of translating pattern-matching bindings and clausal functions into the more rudimentary mechanisms of the internal language. Given a target pattern, a candidate internal-language expression, and a failure exception, the pattern compiler generates a sequence of bindings corresponding to

$$
\begin{array}{l}
[\overline{\texttt{list}} \triangleright \mathit{list} = \mu\, \lambda l{:}\Omega \Rightarrow \Omega.\lambda\alpha{:}\Omega.\Sigma\big(\overline{\texttt{Nil}} \mapsto \mathsf{Unit}, \overline{\texttt{Cons}} \mapsto \alpha \times l\ \alpha\big),\\
\ \mathrm{eq} = \cdots,\\
\ \overline{\texttt{Nil}} = \lambda s{:}[T{:}\Omega].[\mathrm{it} = \mathsf{roll}^{\mathit{list}\ (s.T)}\ (\mathsf{inj}^{\mathsf{Unit}+s.T\times \mathit{list}\ (s.T)}_{\overline{\texttt{Nil}}}\ \{\})],\\
\ \overline{\texttt{Cons}} = \lambda s{:}[T{:}\Omega].[\mathrm{it} = \lambda(x{:}s.T\times \mathit{list}\ (s.T)).\ \mathsf{roll}^{\mathit{list}\ (s.T)}\ (\mathsf{inj}^{\mathsf{Unit}+s.T\times \mathit{list}\ (s.T)}_{\overline{\texttt{Cons}}}\ x)],\\
\ \mathrm{expose} = \lambda s{:}[T{:}\Omega].[\mathrm{it} = \lambda(x{:}\mathit{list}\ (s.T)).\mathsf{unroll}^{\mathit{list}\ (s.T)}\ x]]
\end{array}
$$

Figure 9: The structure $\mathit{mod}_{\texttt{list}}$

the result of matching the candidate against the target. The expected evaluation order is preserved, and an exception is raised if the match does not succeed; these bindings then become the fields of a structure.

Clausal functions are handled by exception propagation (see Appendix D.3). Each clause is compiled into a function that, when applied, matches the argument against the pattern of the clause, and continues with the expression part of the clause in the case that the match succeeds, and fails otherwise. Alternation is handled by generating a function that calls the compilation of the first alternative, yielding its result on success, and passing the argument to the second in the case of failure. Upon failure of the last clause, the internal failure is turned into a `Match` exception. In the case of a `val` binding there is no alternative to failure; a `Bind` exception is raised immediately.

The pattern-compiler given here is unsophisticated, doing sequential search among the patterns and within the patterns until a complete match is found. More efficient algorithms based on decision tree heuristics are routinely used in Standard ML compilers, and using exceptions for chaining function clauses is extremely inefficient. We present a "reference" implementation of pattern compilation so as to avoid undue commitments to specific strategies, to admit generalizations of pattern matching that may engender effects (such as forcing memoized suspensions [20]), and for the sake of perspicuity of the translation.

There is a subtle, but important, interaction between pattern compilation and the value restriction on polymorphism. In the revised *Definition*, the determination of whether or not a variable is generalizable is made based only upon the syntactic form of the right-hand side of a `val` binding (the "value" restriction for polymorphism); it does not matter whether or not the left-hand side is a complex pattern. However, since the pattern match may not succeed, the "true" binding of the variable (after pattern compilation) in a pattern may involve the application of a partial destructuring operation to that value, possibly raising an exception. For example, if `y` is an EL identifier bound to a polymorphic list value (*e.g.*, `Cons(fn x => x, Nil)`) then under Standard ML the binding `val (Cons (x,xs)) = y` will make the variables `x` and `xs` polymorphic since `y` is a syntactic value. In contrast we assess the valuability of the binding of an identifier *after* pattern compilation. Any variable whose "true" binding is not valuable may not be generalized. In particular, the code generated by the pattern compiler will test whether `y` really is a `Cons`, and will raise an exception otherwise. Allowing `y` to be polymorphic would delay any `Bind` exception would be delayed until one of the functors created for `x` or `xs` was instantiated. We therefore do not generalize such identifiers.

However, due to the value restriction, and more generally the definition of total functor, we are guaranteed that a polymorphic value with a sum type has a single fixed tag. In more conventional notation, there is an isomorphism between $\forall\alpha.(\tau_1+\tau_2)$ and $(\forall\alpha.\tau_1)+(\forall\alpha.\tau_2)$. Therefore, one could imagine handling polymorphism for refutable patterns by checking the tag once (by instantiating at some arbitrary type), and either raising a `Bind` exception or using projection from the sum as a total operation on this value thereafter. A weakness of our internal language is that this cannot be expressed; it is unclear how it might be modified to account for this anomaly.

3.6 Equality Compilation

Polymorphic equality, equality type variables, and `eqtype` specifications are all elaborated into explicit uses of equality functions. The idea is to define a canonical equality operation at each closed type, and to associate with each type variable or `eqtype` constructor an equality operation to be supplied by the caller. In the case of equality type variables, polymorphic instantiation provides (passes at run-time) the equality operation based on the instance.[6] In the case of `eqtype` specifications, the signature matching generates the equality test when the signature is ascribed. There is no need for separate "equality attributes" in our IL; a type admits equality if and only if the equality compiler is able to generate an equality operation for it. Our approach is related to the compilation of overloading in Haskell [28] and to the treatment of equality proposed by Gunter, Gunter and MacQueen [6].

The judgment

$$\Gamma \vdash_{\mathrm{eq}} con \rightsquigarrow exp_{\mathrm{v}}$$

expresses that $\Gamma \vdash exp_{\mathrm{v}} : con \times con \rightharpoonup \mathsf{Bool}$ is the equality function for type *con*. These equality functions are the obvious structural equalities for immutable types (primitive equality functions at base types, component-wise equality for record types, a recursively-defined equality function for recursive types, etc.) and primitive pointer equality for reference types.

3.7 Signature Matching

Signature matching is divided into two relations, *instantiation*, which handles type sharing relationships between modules, and *enrichment*, which handles dropping, re-ordering, and instantiation of components of a module. The instantiation relation is captured by the sub-signature relation of the internal language. It is non-coercive in the sense that it has no significance during evaluation. The enrichment relation is handled by the elaboration rules, which introduce coercions that are executed during evaluation. These coercions drop components and introduce polymorphic instantiations to build a structure satisfying a less restrictive signature than that of a given module. Separating the coercive aspects from the internal-language subtyping relation guarantees that the number and order of components in a structure is apparent from its signature.

In one particular case, the coercion introduces, rather than eliminates, components of a structure. This arises because of `eqtype` specifications: the equality compiler must be invoked to determine the appropriate equality function for that type. For example, ascribing the opaque signature

```
sig eqtype T end
```

to a structure having EL signature

```
sig type T = int end
```

must augment the structure containing the type component (equal to Int) with an equality function (equality on integers).

3.8 Type Generativity

One of the more subtle aspects of Standard ML goes under the heading of "type generativity". Roughly speaking, generativity captures the informal idea that a `datatype` declaration introduces a "new" type, distinct from all others, despite possible structural similarities. This aspect of generativity may be regarded as a form of data abstraction. Indeed, in Section 3.4 we relied on opaque signature ascription in the internal language to ensure that the implementation type of a datatype is held abstract.

This basic conception of type generativity must be extended to account for the generative behavior of functors. Datatype generativity interacts with functor instantiation in such a way that each application of a functor that declares a datatype introduces a "new copy" of that datatype, distinct from all other instances introduced by the same functor (and all types otherwise introduced). Following Leroy [13] we capture this behavior by imposing the requirement that module expressions be restricted to "named form". This means that every non-trivial module expression must be bound to a module identifier before it can be used. This restriction is reflected in the grammar by, *e.g.*, the requirement that functor arguments be structure identifiers, rather than arbitrary structure expressions. There is no loss of generality in assuming that programs are written in named form; we can assume a prepass introduces bindings for non-trivial module expression [13]. The practical effect of the restriction to named form is that the result of every functor

[6] One possible alternative would be to add a polymorphic equality primitive to the IL, but the elaborator would still have to do the same bookkeeping to detect during elaboration cases where equality is undefined.

application is bound to module variable, which thereafter serves as the "unique name" of that instance of the functor application. Consequently, opaque types (including datatypes) selected from that instance are unique.

A second subtlety of the Standard ML type system is that types may escape their (apparent) scope. Provided that programs are in named form, this phenomenon can arise in only one way, through the use of `local` declarations and module-level `let` expressions. For example, the following declaration is legal in Standard ML, and results in a binding whose type involves a "hidden" type constructor:

```
local
  datatype t = A | B
in
  val x = A
end
```

The declaration of the datatype `t` is "hidden"; only the variable `x` is exported by the declaration. In our type-theoretic internal language, we clearly cannot allow the binding of `x` to escape the scope of the binding for the type `t`. Instead we export the type `t` along with `x`, but rename it to a variant that lies outside of the "overbar" mapping, ensuring that the type cannot conflict with any user-defined type in the external language.[7] Thus, the "information hiding" of the `local` construct is implemented entirely by the elaborator, and has no significance at the level of the internal language.

A closely-related phenomenon arises in connection with the "transparent ascription" mechanism of Standard ML, whereby signature ascriptions hide components, but not the identities, of types. By hiding a type component that is required to express the type of a value component or the type sharing properties of another type component, we encounter a situation similar to a `local` declaration. For example, in the code

```
functor F(M : sig type 'a con end) : sig type t end =
    struct
        datatype d = D
        type t = d con
    end
```

we would like to express that the type returned by `F` is the result of applying the argument type constructor to a datatype; then we could deduce *solely from the functor's signature* that applying it to the structure

```
struct type 'a con = int end
```

yields a structure containing the type `int`. However, since the transparent ascription "hides" the datatype `d`, we cannot refer to this in describing the returned `t` component. The behavior of this functor on types cannot be expressed in a Standard ML signature.

However, the restriction to named form entails that the ascription generate a module-level `let` expression, which is then translated into type theory by renaming, rather than dropping, the hidden component `d`. Named form and component renaming ensures that the exact behavior of all (first-order) functors is always expressible in the internal-language signature of the translated functor.

Note that the simple renaming mechanism we have outlined here is not "safe for space complexity" [1]. In particular, the elaboration given here retains not only the hidden type components that are required for subsequent specifications, but also type components that are not so required, and value components, which are never required. However, these components may be easily eliminated by a process similar to dead code elimination in a compiler. In practice we would retain only those hidden type components that are necessary to ensure that the translation is well-formed.

3.9 Properties of the Elaborator

The minimal requirement for the elaborator is that the elaboration of external-language code yields well-formed internal-language code:

[7] More precisely, the "hidden" part of a `local` declaration is represented by a substructure with an inaccessible name, and the references to hidden identifiers are replaced by accesses to the substructure. Conflicts are avoided by α-conversion of the internal name of the structure.

Proposition 6 (Well-formed translation)
If $\vdash \Gamma$ *ok and* $\Gamma \vdash$ *EL-phrase* $\rightsquigarrow$ *phrase* : *class then* $\Gamma \vdash$ *phrase* : *class.*

The elaboration rules in the Appendix D assume a structure variable *basis* which represents the initial basis for programs. For our purposes here, it suffices to assume a structure with signature sig_{basis}, given by

$$\begin{array}{ll}
[\overline{\mathtt{Bind}}^* & :[\mathrm{tag}{:}\mathsf{Unit\ Tag}, \overline{\mathtt{Bind}}{:}\mathsf{Tagged}], \\
\overline{\mathtt{Match}}^* & :[\mathrm{tag}{:}\mathsf{Unit\ Tag}, \overline{\mathtt{Match}}{:}\mathsf{Tagged}], \\
\mathrm{fail}^* & :[\mathrm{tag}{:}\mathsf{Unit\ Tag}, \mathrm{fail}{:}\mathsf{Tagged}], \\
\overline{\mathtt{bool}}^* & :[\overline{\mathtt{bool}} \triangleright b{:}\Omega{=}\Sigma(\mathsf{Unit}, \mathsf{Unit}), \\
& \quad \mathrm{eq}{:}b \times b \rightharpoonup \mathsf{Bool}, \\
& \quad \overline{\mathtt{false}}{:}b, \\
& \quad \overline{\mathtt{true}}{:}b, \\
& \quad \mathrm{expose}{:}b \rightarrow \Sigma(\mathsf{Unit}, \mathsf{Unit})].
\end{array}$$

The interpretation of Standard ML we have outlined above relies on a relational presentation of what is essentially a translation function. The relational framework allows us to avoid overspecifying the translation, and admits a clean separation between "algorithmic" and "definitional" considerations. However, we incur the obligation to demonstrate that the interpretation is *coherent* in the sense that all interpretations of a Standard ML program yield internal-language expressions with the same observable behavior. We conjecture that the translation we have given is coherent:

Conjecture 7 (Coherence)
If

$$\mathrm{basis}^* \triangleright basis{:}sig_{basis} \vdash expr \rightsquigarrow exp : ans$$

and

$$\mathrm{basis}^* \triangleright basis{:}sig_{basis} \vdash expr \rightsquigarrow exp' : ans$$

and

$$(\cdot, \cdot, \mathsf{let}\ basis{=}mod_{basis}\ \mathsf{in}\ exp\ \mathsf{end}) \hookrightarrow^* \Sigma_t$$

then for some terminal state $\Sigma'_t \cong \Sigma_t$,

$$(\cdot, \cdot, \mathsf{let}\ basis{=}mod_{basis}\ \mathsf{in}\ exp'\ \mathsf{end}) \hookrightarrow^* \Sigma'_t.$$

4 Summary

We have given a brief overview of an interpretation of Standard ML into a kernel type theory. A fully detailed account of the interpretation appears in a companion technical report [10]. The complete interpretation consists of approximately 270 inference rules, of which approximately 140 form the typing rules and dynamic semantics of the internal language (120 rules and 20 rules, respectively), with the remaining 130 rules being part of the interpretation itself. Of these approximately 10 rules are concerned with signature matching, 8 with equality compilation, 25 with identifier lookup, and 12 with pattern compilation. By contrast *The Definition of Standard ML* consists of approximately 190 rules, of which approximately 100 are for the static semantics, the remainder being for the dynamic semantics. Note, however, that the dynamic semantics has "implicit" rules for handling exceptions, making it difficult to give a precise count.

Our internal language is formalized using standard techniques. The type checking rules rely on conventions such as implicit α-conversion of binding operators to avoid identifier conflicts, and relies on definitional equality relations and a sub-typing relations to define the type system. The operational semantics is defined by a transition relation on states of an abstract machine, and does not rely on implicit rules for exception propagation. It can be easily extended to account for control operators such as call-with-current-continuation. The internal language admits a clean formulation of the soundness theorem that does not rely on instrumentation of the rules with explicit "wrong" transitions. To properly state soundness in the framework of *The Definition* requires that the dynamic semantics be instrumented with explicit "wrong" transitions, which would significantly increase the number of rules required to give a rigorous formulation of the language comparable to the one given here. Finally, we note that the internal language does not rely on any global "admissibility" conditions as are imposed on the static semantic objects of *The Definition*.

The translation from Standard ML into the internal language is, at times, rather complex. The single most complicated rule — for handling datatype declarations — requires one page in the technical report. The complexity is easily explained: a single datatype declaration introduces n mutually recursive type constructors, each with its own arity, and each introducing k_i value constructors, each of which may or may not take an argument. Unravelling these complexities into the simple orthogonal mechanisms of the internal language is clearly a rather complicated affair. Other sources of complexity are the use of rules to define identifier lookup and signature "patching", the introduction of coercions for signature matching, the compilation of "equality types" into modules consisting of a type and an equality operation, and the compilation of patterns into primitive projections.

How might the presentation be simplified? The use of rules for identifier lookup and signature specialization is a matter of presentation. We could easily have defined these at the metalevel of the semantics, rather than give explicit rules. Equality compilation introduces considerable complexity. Since we are working in an explicitly-typed framework we could have postulated in the internal language a primitive polymorphic equality operation that dispatches on types. We chose not to do so primarily because the elaborator would nevertheless have to check for admissibility of equality at compile time to ensure that invalid uses of equality are rejected during type checking. It is only marginally more complicated to equip equality types with their equality operation and eliminate equality checking from the internal language entirely. We see no plausible alternative to the coercive interpretation of signature matching. One might consider enriching the internal language with a coercive pre-order on signatures corresponding to the enrichment ordering, but to do so would require unnatural, ML-specific extensions such as implicit polymorphic instantiation during signature matching. The treatment of datatypes and pattern matching appears to be essentially forced, given that in an explicitly-typed framework it is necessary to break abstraction during evaluation to expose the underlying representation of a datatype. More generally, the association between an abstract type and its representation must be made explicit in the dynamic semantics, and this is precisely what is accomplished here. We consider it an important direction for further research to determine if a simpler treatment of datatypes can be given in an explicitly-typed framework.

The interpretation we have given here follows closely the model of *The Definition* in clearly separating definitional from algorithmic issues. The rules exploit the indeterminacy of the relational framework for the sake of simplicity and concision. The internal language type system is used to express context-sensitive formation constraints. An implementation must resolve these indeterminacies and must define algorithms for the internal language type system. A thorough treatment of these matters lies beyond the scope of this work. We remark, however, that the elaboration relation is decidable, and has been implemented in the TIL/ML compiler. The elaborator relies on the existence of principal signatures for modules, which exist for precisely the same reasons as in *The Definition*.[8] The type checking conditions that arise during elaboration are all decidable.

The type-theoretic interpretation has both advantages and disadvantages as compared to *The Definition*. The primary disadvantage is that the dynamic semantics of Standard ML must be understood by translation into the internal language. Since the translation rules are not fully determinate, this raises the question of coherence of the translation, which we conjecture to hold for the translation given here. There is also the question the psychological question of whether the kind of translation we give here can serve as an alternative to *The Definition* as a reference for programmers. Here there is good argument for the simpler methods of *The Definition*. However, it seems to us that few people learn Standard ML by reading the formal definition. Thus the relative merits of the two approaches seem unclear. As a tool for compiler-writers, both *The Definition* and the interpretation we propose here have contributed directly to the construction of practical implementations of Standard ML. In this regard the two accounts complement one another — different compiler technologies require different interpretations of the language.

[8]Note, however, that principal signatures are not necessary for formulating the translation since we do not rely on "generic structures" for a given signature to define the interpretation.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[3] Luca Cardelli. Type systems. In Allen B. Tucker Jr., editor, *Handbook of Computer Science and Engineering*, pages 2208–2236. CRC Press, 1997.

[4] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, 1971.

[5] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: a mechanised logic of computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[6] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. An abstract interpretation for ML equality kinds. *Lecture Notes in Computer Science*, 526:112–130, 1991.

[7] Robert Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, School of Computer Science, Carnegie Mellon University, 1993.

[8] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[9] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[10] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (Version 2). Technical report, School of Computer Science, Carnegie Mellon University. Forthcoming.

[11] Xavier Leroy. Polymorphism by name for references and continuations. In *20th ACM Symposium on Principles of Programming Languages*, pages 220–231, 1993.

[12] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[13] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[14] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[15] David MacQueen. Using dependent types to express modular structure. In *13th ACM Symposium on Principles of Programming Languages*, pages 277–286, 1986.

[16] Robin Milner. A proposal for Standard ML. In *1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[17] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. The Definition of Standard ML, Revised 1996. Forthcoming.

[18] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[19] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[20] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[21] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proceedings of the 18th International Symposium*, volume 711 of *LNCS*, pages 122–141. Springer-Verlag, Berlin, 1993.

[22] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Aarhus University, September 1981.

[23] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium, Proceedings, Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425, 1974.

[24] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, September 1976.

[25] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ml. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, 1995.

[26] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[27] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

[28] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[29] Andrew Wright. Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation*, 8(4):343–355, December 1995.

[30] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Dept. of Computer Science, Rice University, 1991.

A External Language Syntax

A.1 Abstract Syntax

$expr ::= scon$
$\mid longid$
$\mid \texttt{\{}lab_1 \texttt{ = } expr_1\texttt{,} \cdots \texttt{,} lab_n \texttt{ = } expr_n\texttt{\}}$
$\mid \texttt{let } strdec \texttt{ in } expr \texttt{ end}$
$\mid expr\ expr'$
$\mid expr \texttt{ : } ty$
$\mid expr \texttt{ handle } match$
$\mid \texttt{raise } expr$
$\mid \texttt{fn } match$
$\mid expr_1 \texttt{ = } expr_2$

$mrule ::= pat \texttt{ => } expr$
$match ::= mrule$
$\mid mrule \texttt{ | } match$

$strdec ::= \cdot$
$\mid \texttt{val (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } pat \texttt{ = } exp$
$\mid \texttt{val (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) rec } pat \texttt{ = } exp$
$\mid strdec_1\ strdec_2$
$\mid \texttt{open } longid_1 \cdots longid_n$
$\mid \texttt{exception } id$
$\mid \texttt{exception } id \texttt{ of } ty$
$\mid \texttt{exception } id \texttt{ = } longid$
$\mid \texttt{local } strdec_1 \texttt{ in } strdec_2 \texttt{ end}$
$\mid \texttt{type } tybind$
$\mid \texttt{datatype } datbind$
$\mid \texttt{datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } tycon \texttt{ =}$
$\quad \texttt{datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } longtycon$
$\mid \texttt{structure } strbind$
$\mid \texttt{functor } funbind$

$tybind ::= \langle\texttt{(}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{)}\rangle\ tycon \texttt{ = } ty\ \langle\langle\texttt{and } tybind\rangle\rangle$
$datbind ::= \texttt{(}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } tycon \texttt{ = } conbind$
$\quad \langle\texttt{and } datbind\rangle$
$conbind ::= id\ \langle\texttt{of } ty\rangle\ \langle\langle\texttt{| } conbind\rangle\rangle$

$strexp ::= longstrid$
$\mid \texttt{struct } strdec \texttt{ end}$
$\mid longfunid \texttt{ (}longstrid\texttt{)}$
$\mid longstrid \texttt{ : } sigexp$
$\mid longstrid \texttt{ :> } sigexp$
$\mid \texttt{let } strdec \texttt{ in } strexp \texttt{ end}$

$$
\begin{aligned}
\mathit{spec} ::= {} & \cdot \\
& \mid \texttt{val}\ \mathit{id}\ \texttt{:}\ \mathit{ty} \\
& \mid \texttt{type}\ \mathit{typdesc} \\
& \mid \texttt{eqtype}\ \mathit{etypdesc} \\
& \mid \texttt{datatype}\ \mathit{datbind} \\
& \mid \texttt{datatype}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{tycon}'\ \texttt{=} \\
& \quad \texttt{datatype}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{longtycon} \\
& \mid \texttt{exception}\ \mathit{id} \\
& \mid \texttt{exception}\ \mathit{id}\ \texttt{of}\ \mathit{ty} \\
& \mid \texttt{structure}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp} \\
& \mid \texttt{functor}\ \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{:}\ \mathit{sigexp}' \\
& \mid \texttt{include}\ \mathit{sigexp} \\
& \mid \mathit{spec}_1\ \mathit{spec}_2 \\
& \mid \mathit{spec}\ \texttt{sharing type}\ \mathit{longid}_1\ \texttt{=}\ \mathit{longid}_2
\end{aligned}
$$

$$
\begin{aligned}
\mathit{typdesc} ::= {} & \langle\texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{tycon} \\
& \quad \langle\langle\texttt{and}\ \mathit{typdesc}\rangle\rangle \\
& \mid \langle\texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{tycon}\ \texttt{=}\ \mathit{ty} \\
& \quad \langle\langle\texttt{and}\ \mathit{typdesc}\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{etypdesc} ::= {} & \langle\texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{tycon} \\
& \quad \langle\langle\texttt{and}\ \mathit{etypdesc}\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{sigexp} ::= {} & \texttt{sig}\ \mathit{spec}\ \texttt{end} \\
& \mid \mathit{sigexp}\ \texttt{where type} \\
& \quad \langle\texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{pat} ::= {} & \mathit{scon} \\
& \mid \mathit{longid} \\
& \mid \texttt{\_} \\
& \mid \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
& \mid \mathit{longid}\ \mathit{pat} \\
& \mid \texttt{\{}\mathit{lab}_1\ \texttt{=}\ \mathit{pat}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n\ \texttt{=}\ \mathit{pat}_n\langle\texttt{,}\ \texttt{...}\rangle\texttt{\}} \\
& \mid \mathit{pat}_1\ \texttt{as}\ \mathit{pat}_2 \\
& \mid \texttt{ref}\ \mathit{pat}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ty} ::= {} & \mathit{base} \\
& \mid \mathit{tyvar} \\
& \mid \texttt{\{}\mathit{lab}_1\ \texttt{:}\ \mathit{expr}_1\texttt{,}\cdots\texttt{,}\mathit{lab}_n\ \texttt{:}\ \mathit{expr}_n\texttt{\}} \\
& \mid \langle\texttt{(}\mathit{ty}_1\texttt{,}\cdots\texttt{,}\mathit{ty}_n\texttt{)}\rangle\ \mathit{longtycon} \\
& \mid \mathit{ty}\ \texttt{->}\ \mathit{ty}'
\end{aligned}
$$

$$
\begin{aligned}
\mathit{strbind} ::= {} & \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{strbind}\rangle \\
\mathit{funbind} ::= {} & \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{=}\ \mathit{strexp} \\
& \quad \langle\texttt{and}\ \mathit{funbind}\rangle
\end{aligned}
$$

This grammar has a few minor differences from that specified in the revised *Definition*. We have simplified the grammar by removing some of the distinctions made solely for the purposes of the parser, which are inappropriate for abstract syntax. We extend the grammar to allow for module definitions local to an expression, and for functor specifications in signatures. We do not support `abstype` here; in the presence of local module definitions and the opaque (`:>`) signature ascription, `abstype` is redundant. For simplicity, we assume that `signature` declarations are syntactic sugar which have been "inlined away." See the *Definition* for further syntactic restrictions information on how derived forms desugar into the above grammar. As in the Definition, we use the convention that angle brackets and double angle brackets mark optional components of a rule or syntactic item.

B Internal Language Static Semantics (excerpt)

B.1 Constructor Equivalence

$$\frac{\vdash \mathit{decs}\ \text{ok} \qquad \mathit{decs} = \mathit{decs}', \mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{decs}''}{\mathit{decs} \vdash \mathit{var} \equiv \mathit{con} : \mathit{knd}} \tag{1}$$

$$\frac{\mathit{decs} \vdash \mathit{mod}_{\mathrm{v}} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs}'] \qquad \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{con}) = \emptyset}{\mathit{decs} \vdash \mathit{mod}_{\mathrm{v}}.\mathit{lab} \equiv \mathit{con} : \mathit{knd}} \tag{2}$$

$$\frac{\mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \mathit{decs} \vdash \mathit{con}_1' \equiv \mathit{con}_2' : \Omega}{\mathit{decs} \vdash \mathit{con}_1 \rightharpoonup \mathit{con}_1' \equiv \mathit{con}_2 \rightharpoonup \mathit{con}_2' : \Omega} \tag{3}$$

$$\frac{\mathit{decs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \mathit{decs} \vdash \mathit{con}_1' \equiv \mathit{con}_2' : \Omega}{\mathit{decs} \vdash \mathit{con}_1 \rightarrow \mathit{con}_1' \equiv \mathit{con}_2 \rightarrow \mathit{con}_2' : \Omega} \tag{4}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{con}\ \mathsf{Ref} \equiv \mathit{con}'\ \mathsf{Ref} : \Omega} \tag{5}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{con}\ \mathsf{Tag} \equiv \mathit{con}'\ \mathsf{Tag} : \Omega} \tag{6}$$

$$\frac{\mathit{lab}_1, \ldots, \mathit{lab}_n\ \text{distinct} \qquad \forall i \in 1..n : \quad \mathit{decs} \vdash \mathit{con}_i \equiv \mathit{con}_i' : \Omega}{\mathit{decs} \vdash \{\mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n\} \equiv \{\mathit{lab}_1{:}\mathit{con}_1', \cdots, \mathit{lab}_n{:}\mathit{con}_n'\} : \Omega} \tag{7}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \Rightarrow \mathit{knd}}{\mathit{decs} \vdash \mu\, \mathit{con} \equiv \mu\, \mathit{con}' : \mathit{knd}} \tag{8}$$

$$\frac{\langle i \in 1..n\rangle \qquad \forall i \in 1..n : \quad \mathit{decs} \vdash \mathit{con}_i \equiv \mathit{con}_i' : \Omega}{\mathit{decs} \vdash \Sigma_{\langle \mathit{lab}_i\rangle}\, (\mathit{lab}_1 \mapsto \mathit{con}_1, \ldots, \mathit{lab}_n \mapsto \mathit{con}_n) \equiv \Sigma_{\langle \mathit{lab}_i\rangle}\, (\mathit{lab}_1 \mapsto \mathit{con}_1', \ldots, \mathit{lab}_n \mapsto \mathit{con}_n') : \Omega} \tag{9}$$

$$\frac{\begin{array}{c} decs \vdash con_1 \equiv con_2 : knd' \Rightarrow knd \\ decs \vdash con_1' \equiv con_2' : knd' \end{array}}{decs \vdash con_1\ con_2 \equiv con_1'\ con_2' : knd} \quad (10)$$

$$\frac{decs, var{:}knd' \vdash con : knd \qquad decs \vdash con : knd'}{decs \vdash (\lambda var{:}knd'.con)\ con' \equiv \{con'/var\}con : knd} \quad (11)$$

$$\frac{decs \vdash con : knd}{decs \vdash con \equiv con : knd} \quad (12)$$

$$\frac{decs \vdash con' \equiv con : knd}{decs \vdash con \equiv con' : knd} \quad (13)$$

$$\frac{\begin{array}{c} decs \vdash con \equiv con' : knd \\ decs \vdash con' \equiv con'' : knd \end{array}}{decs \vdash con \equiv con'' : knd} \quad (14)$$

B.2 Well-formed Expressions

$$\frac{\begin{array}{c} \vdash decs \text{ ok} \\ decs = decs', var{:}con, decs'' \end{array}}{decs \vdash var : con} \quad (15)$$

$$\frac{\begin{array}{c} \vdash decs \text{ ok} \\ decs = decs', loc{:}con, decs'' \end{array}}{decs \vdash loc : con} \quad (16)$$

$$\frac{\begin{array}{c} \vdash decs \text{ ok} \\ decs = decs', tag{:}con, decs'' \end{array}}{decs \vdash tag : con} \quad (17)$$

$$\frac{decs \vdash exp : con' \rightharpoonup con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \quad (18)$$

$$\frac{decs \vdash exp : con' \rightarrow con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \quad (19)$$

$$\frac{\begin{array}{l} \forall i \in 1..n : \\ \quad decs, (var_j'{:}con_j \rightharpoonup con_j')_{j=1}^{n}, var_i{:}con_i \vdash exp_i : con_i' \end{array}}{\begin{array}{c} decs \vdash \mathsf{fix}\ (var_i'(var_i{:}con_i){:}con_i' \mapsto exp_i)_{i=1}^{n}\ \mathsf{end} : \\ \{1{=}con_1 \rightharpoonup con_1', \ldots, n{=}con_n \rightharpoonup con_n'\} \end{array}} \quad (20)$$

$$\frac{\begin{array}{c} var' \notin \mathrm{FV}\,exp \\ decs, var{:}con \vdash exp \downarrow con' \end{array}}{\begin{array}{c} decs \vdash \mathsf{fix}\ var'(var{:}con){:}con' \mapsto exp\ \mathsf{end} : \\ \{\overline{1}{=}con \rightarrow con'\} \end{array}} \quad (21)$$

$$\frac{\begin{array}{c} lab_1, \cdots, lab_n \text{ distinct} \\ \forall i \in 1..n : \quad decs \vdash exp_i : con_i \end{array}}{\begin{array}{c} decs \vdash \{lab_1{=}exp_1, \cdots, lab_n{=}exp_n\} : \\ \{lab_1{:}con_1, \cdots, lab_n{:}con_n\} \end{array}} \quad (22)$$

$$\frac{decs \vdash exp : \{rdecs, lab{:}con, rdecs'\}}{decs \vdash \pi_{lab}\ exp : con} \quad (23)$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp' : \mathsf{Tagged} \rightharpoonup con}{decs \vdash \mathsf{handle}\ exp\ \mathsf{with}\ exp' : con} \quad (24)$$

$$\frac{decs \vdash exp : \mathsf{Tagged} \qquad decs \vdash con : \Omega}{decs \vdash \mathsf{raise}^{con}\ exp : con} \quad (25)$$

$$\frac{decs \vdash con : \Omega}{decs \vdash \mathsf{new\_tag}[con] : con\ \mathsf{Tag}} \quad (26)$$

$$\frac{decs \vdash exp : con}{decs \vdash \mathsf{ref}^{con}\ exp : con\ \mathsf{Ref}} \quad (27)$$

$$\frac{decs \vdash exp : con\ \mathsf{Ref}}{decs \vdash \mathsf{get}\ exp : con} \quad (28)$$

$$\frac{\begin{array}{c} decs \vdash exp : con\ \mathsf{Ref} \\ decs \vdash exp' : con \end{array}}{decs \vdash \mathsf{set}\ (exp, exp') : \mathsf{Unit}} \quad (29)$$

$$\frac{\begin{array}{c} decs \vdash con \equiv (\pi_{lab}\ (\mu\ con'))\ \langle con'' \rangle : \Omega \\ decs \vdash exp : (\pi_{lab}\ (con'\ (\mu\ con')))\ \langle con'' \rangle \end{array}}{decs \vdash \mathsf{roll}^{con}\ exp : con} \quad (30)$$

$$\frac{\begin{array}{c} decs \vdash con \equiv (\pi_{lab}\ (\mu\ con'))\ \langle con'' \rangle : \Omega \\ decs \vdash exp : con \end{array}}{decs \vdash \mathsf{unroll}^{con}\ exp : (\pi_{lab}\ (con'\ (\mu\ con')))\ \langle con'' \rangle} \quad (31)$$

$$\frac{decs \vdash exp : con \rightarrow con'}{decs \vdash \partial\ exp : con \rightharpoonup con'} \quad (32)$$

$$\frac{\begin{array}{c} i \in 1..n \\ con = \Sigma_{lab_i}\ (lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) \\ decs \vdash exp : con_i \end{array}}{decs \vdash \mathsf{inj}_{lab_i}^{con}\ exp : con} \quad (33)$$

$$\frac{\begin{array}{c} i \in 1..n \\ decs \vdash exp : \Sigma_{lab_i}\ (lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) \end{array}}{decs \vdash \mathsf{proj}_{lab_i}^{\Sigma_{lab_i}(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n)}\ exp : con_i} \quad (34)$$

$$\frac{\begin{array}{c} n \geq 1 \\ \mathit{con} = \Sigma(\mathit{lab}_1{\mapsto}\mathit{con}_1, \ldots, \mathit{lab}_n{\mapsto}\mathit{con}_n) \\ \mathit{decs} \vdash \mathit{exp} : \mathit{con} \\ \forall i \in 1..n : \\ \mathit{decs} \vdash \mathit{exp}_i : \Sigma_{\mathit{lab}_i}(\mathit{lab}_1{\mapsto}\mathit{con}_1, \ldots, \mathit{lab}_n{\mapsto}\mathit{con}_n) \rightharpoonup \mathit{con}' \end{array}}{\mathit{decs} \vdash \mathsf{case}^{\mathit{con}}\ \mathit{exp}\ \mathsf{of}\ \mathit{exp}_1, \ldots, \mathit{exp}_n\ \mathsf{end} : \mathit{con}'} \tag{35}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}\ \mathsf{Tag} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}}{\mathit{decs} \vdash \mathsf{tag}(\mathit{exp}, \mathit{exp}') : \mathsf{Tagged}} \tag{36}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{exp} : \mathsf{Tagged} \qquad \mathit{decs} \vdash \mathit{exp}' : \mathit{con}\ \mathsf{Tag} \\ \mathit{decs} \vdash \mathit{exp}'' : \mathit{con} \rightharpoonup \mathit{con}' \qquad \mathit{decs} \vdash \mathit{exp}''' : \mathit{con}' \end{array}}{\mathit{decs} \vdash \mathsf{iftagof}\ \mathit{exp}\ \mathsf{is}\ \mathit{exp}'\ \mathsf{then}\ \mathit{exp}''\ \mathsf{else}\ \mathit{exp}''' : \mathit{con}'} \tag{37}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{con}, \mathit{sdecs}'] \\ \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{con}) = \emptyset \end{array}}{\mathit{decs} \vdash \mathit{mod}.\mathit{lab} : \mathit{con}} \tag{38}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}' \qquad \mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\mathit{decs} \vdash \mathit{exp} : \mathit{con}} \tag{39}$$

B.3 Signature Subtyping

$$\frac{}{\mathit{decs} \vdash \cdot \ \leq\ \cdot} \tag{40}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd}{=}\mathit{con} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs}\ \leq\ \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}'} \tag{41}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig} \\ \mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}' \end{array}}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs}\ \leq\ \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'} \tag{42}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{lab}{:}\mathit{dec}\ \equiv\ \mathit{lab}{:}\mathit{dec}' \\ \mathit{decs}, \mathit{dec} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}' \end{array}}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{dec}, \mathit{sdecs}\ \leq\ \mathit{lab}{\triangleright}\mathit{dec}', \mathit{sdecs}'} \tag{43}$$

$$\frac{\mathit{decs} \vdash \mathit{sdecs}\ \leq\ \mathit{sdecs}'}{\mathit{decs} \vdash [\mathit{sdecs}] \leq [\mathit{sdecs}'] : \mathsf{Sig}} \tag{44}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \\ \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig} \end{array}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \rightharpoonup \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 \rightharpoonup \mathit{sig}_2' : \mathsf{Sig}} \tag{45}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \to \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 \rightharpoonup \mathit{sig}_2' : \mathsf{Sig}} \tag{46}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 \to \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 \to \mathit{sig}_2' : \mathsf{Sig}} \tag{47}$$

B.4 Well-formed Modules

$$\frac{}{\mathit{decs} \vdash \cdot : \cdot} \tag{48}$$

$$\frac{\mathit{decs} \vdash \mathit{bnd}\ :\ \mathit{dec} \qquad \mathit{decs}, \mathit{dec} \vdash \mathit{sbnds}\ :\ \mathit{sdecs}}{\mathit{decs} \vdash \mathit{lab}{\triangleright}\mathit{bnd}, \mathit{sbnds}\ :\ \mathit{lab}{\triangleright}\mathit{dec}, \mathit{sdecs}} \tag{49}$$

$$\frac{\begin{array}{c} \vdash \mathit{decs}\ \mathrm{ok} \\ \mathit{decs} = \mathit{decs}', \mathit{var}{:}\mathit{sig}, \mathit{decs}'' \end{array}}{\mathit{decs} \vdash \mathit{var} : \mathit{sig}} \tag{50}$$

$$\frac{\mathit{decs} \vdash \mathit{sbnds}\ :\ \mathit{sdecs}}{\mathit{decs} \vdash [\mathit{sbnds}] : [\mathit{sdecs}]} \tag{51}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{mod} : \mathit{sig}'}{\mathit{decs} \vdash \lambda \mathit{var}{:}\mathit{sig}.\mathit{mod} : \mathit{var}{:}\mathit{sig} \rightharpoonup \mathit{sig}'} \tag{52}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{mod} \downarrow \mathit{sig}'}{\mathit{decs} \vdash \lambda \mathit{var}{:}\mathit{sig}.\mathit{mod} : \mathit{var}{:}\mathit{sig} \to \mathit{sig}'} \tag{53}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}' \rightharpoonup \mathit{sig} \qquad \mathit{decs} \vdash \mathit{mod}' : \mathit{sig}'}{\mathit{decs} \vdash \mathit{mod}\ \mathit{mod}' : \mathit{sig}} \tag{54}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod} : [\mathit{sdecs}, \mathit{lab}{:}\mathit{sig}, \mathit{sdecs}'] \\ \mathrm{BV}(\mathit{sdecs}) \cap \mathrm{FV}(\mathit{sig}) = \emptyset \end{array}}{\mathit{decs} \vdash \mathit{mod}.\mathit{lab} : \mathit{sig}} \tag{55}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}}{\mathit{decs} \vdash \mathit{mod}{:}\mathit{sig} : \mathit{sig}} \tag{56}$$

$$\frac{\mathit{decs} \vdash \mathit{mod}_{\mathsf{v}}\ : [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}']}{\mathit{decs} \vdash \mathit{mod}_{\mathsf{v}}\ : [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{mod}_{\mathsf{v}}.\mathit{lab}, \mathit{sdecs}']} \tag{57}$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{mod}_{\mathsf{v}}\ : [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs}'] \\ \mathit{decs} \vdash \mathit{mod}_{\mathsf{v}}.\mathit{lab} : \mathit{sig}' \end{array}}{\mathit{decs} \vdash \mathit{mod}_{\mathsf{v}}\ : [\mathit{sdecs}, \mathit{lab}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}']} \tag{58}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig} \qquad \mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}'} \tag{59}$$

B.5 Valuability Judgments

$$\frac{decs \vdash exp : con \qquad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \tag{60}$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \tag{61}$$

$$\frac{}{decs \vdash exp_{\mathrm{v}} \downarrow} \tag{62}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \tag{63}$$

$$\frac{decs \vdash exp_1 \downarrow con' \to con \qquad decs \vdash exp_2 \downarrow}{decs \vdash exp_1\; exp_2 \downarrow} \tag{64}$$

$$\frac{decs \vdash exp_1 \downarrow \qquad \cdots \qquad decs \vdash exp_n \downarrow}{decs \vdash \{lab_1{=}exp_1, \cdots, lab_n{=}exp_n\} \downarrow} \tag{65}$$

$$\frac{}{decs \vdash mod_{\mathrm{v}} \downarrow} \tag{66}$$

$$\frac{decs \vdash mod \downarrow sig' \to sig \qquad decs \vdash mod' \downarrow}{decs \vdash mod\; mod' \downarrow} \tag{67}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \tag{68}$$

C IL Dynamic Semantics

$$\begin{array}{l}(\Delta, \sigma, E[(\langle\partial\rangle \pi_{\overline{k}}\, exp_{\mathrm{v}})\; exp_{\mathrm{v}}{}'])\hookrightarrow \\ \quad (\Delta, \sigma, E[\{\pi_{\overline{1}}\, exp_{\mathrm{v}}/var_1\}\cdots\{\pi_{\overline{n}}\, exp_{\mathrm{v}}/var_n\}\{exp_{\mathrm{v}}{}'/var'_k\}exp_k]) \\ \text{where} \quad exp_{\mathrm{v}} = \mathsf{fix}\,(var_i(var'_i{:}con_i){:}con'_i \mapsto exp_i)_{i=1}^{n}\;\mathsf{end}\end{array} \tag{69}$$

$$\begin{array}{l}(\Delta, \sigma, E[\pi_{lab}\,\{rbnds_{\mathrm{v}}, lab{=}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}{}'\}])\hookrightarrow \\ \quad (\Delta, \sigma, E[exp_{\mathrm{v}}])\end{array} \tag{70}$$

$$(\Delta, \sigma, E[\mathsf{handle}\; exp_{\mathrm{v}}\;\mathsf{with}\; exp]) \hookrightarrow (\Delta, \sigma, E[exp_{\mathrm{v}}]) \tag{71}$$

$$\begin{array}{l}(\Delta, \sigma, E[\mathsf{ref}^{con}\; exp_{\mathrm{v}}]) \hookrightarrow (\Delta[loc{:}con], \sigma[loc \mapsto exp_{\mathrm{v}}], E[loc]) \\ \quad \text{if } loc \notin \mathrm{BV}(\Delta)\end{array} \tag{72}$$

$$(\Delta, \sigma, E[\mathsf{get}\; loc]) \hookrightarrow (\Delta, \sigma, E[\sigma(loc)]) \tag{73}$$

$$(\Delta, \sigma, E[\mathsf{set}\,(loc, exp_{\mathrm{v}})]) \hookrightarrow (\Delta, \sigma[loc \mapsto exp_{\mathrm{v}}], E[\{\}]) \tag{74}$$

$$(\Delta, \sigma, E[\mathsf{unroll}^{con}\;(\mathsf{roll}^{con'}\; exp_{\mathrm{v}})]) \hookrightarrow (\Delta, \sigma, E[exp_{\mathrm{v}}]) \tag{75}$$

$$\begin{array}{l}(\Delta, \sigma, E[\mathsf{new\_tag}[con]]) \hookrightarrow (\Delta[tag{:}con\;\mathsf{Tag}], \sigma, E[tag]) \\ \quad \text{if } tag \notin \mathrm{BV}(\Delta)\end{array} \tag{76}$$

$$(\Delta, \sigma, E[\mathsf{proj}\,(\mathsf{inj}_i^{con}\; exp_{\mathrm{v}})]) \hookrightarrow (\Delta, \sigma, E[exp_{\mathrm{v}}]) \tag{77}$$

$$\begin{array}{l}(\Delta, \sigma, E[\mathsf{iftagof}\;\mathsf{tag}(tag, exp_{\mathrm{v}})\;\mathsf{is}\; tag\;\mathsf{then}\; exp\;\mathsf{else}\; exp']) \hookrightarrow \\ \quad (\Delta, \sigma, E[exp\; exp_{\mathrm{v}}])\end{array} \tag{78}$$

$$\begin{array}{l}(\Delta, \sigma, E[\mathsf{iftagof}\;\mathsf{tag}(tag', exp_{\mathrm{v}})\;\mathsf{is}\; tag\;\mathsf{then}\; exp\;\mathsf{else}\; exp']) \hookrightarrow \\ \quad (\Delta, \sigma, E[exp']) \\ \text{if } tag \neq tag'\end{array} \tag{79}$$

$$\begin{array}{l}(\Delta, \sigma, E[(\lambda var{:}sig.mod)\; mod_{\mathrm{v}}]) \hookrightarrow \\ \quad (\Delta, \sigma, E[\{mod_{\mathrm{v}}/var\}mod])\end{array} \tag{80}$$

$$(\Delta, \sigma, E[mod_{\mathrm{v}}{:}sig]) \hookrightarrow (\Delta, \sigma, E[mod_{\mathrm{v}}]) \tag{81}$$

$$\begin{array}{l}(\Delta, \sigma, E[[sbnds_{\mathrm{v}}, lab{=}val, sbnds_{\mathrm{v}}{}'].lab]) \hookrightarrow \\ \quad (\Delta, \sigma, E[val]) \\ \qquad\qquad \text{where } \mathrm{BV}(sbnds_{\mathrm{v}}) \cap \mathrm{FV}(val) = \emptyset\end{array} \tag{82}$$

$$\begin{array}{l}(\Delta, \sigma, E[exp_{\mathrm{v}} =_{con} exp_{\mathrm{v}}{}']) \hookrightarrow (\Delta, \sigma, E[\mathsf{true}]) \\ \quad \text{if } exp_{\mathrm{v}} \text{ and } exp_{\mathrm{v}}{}' \text{ are equal at type } con\end{array} \tag{83}$$

$$\begin{array}{l}(\Delta, \sigma, E[exp_{\mathrm{v}} =_{con} exp_{\mathrm{v}}{}']) \hookrightarrow (\Delta, \sigma, E[\mathsf{false}]) \\ \quad \text{if } exp_{\mathrm{v}} \text{ and } exp_{\mathrm{v}} \text{ are unequal at type } con\end{array} \tag{84}$$

$$\begin{array}{l}(\Delta, \sigma, E[\mathsf{handle}\; R[\mathsf{raise}^{con}\; exp_{\mathrm{v}}]\;\mathsf{with}\; exp']) \hookrightarrow \\ \quad (\Delta, \sigma, E[exp'\; exp_{\mathrm{v}}])\end{array} \tag{85}$$

$$\begin{array}{l}(\Delta, \sigma, R[\mathsf{raise}^{con}\; exp_{\mathrm{v}}]) \hookrightarrow (\Delta, \sigma, \mathsf{raise}^{ans}\; exp_{\mathrm{v}}) \\ \quad \text{if } R \neq []\end{array} \tag{86}$$

D Elaboration (excerpts)

D.1 Derived forms

$knd_1 \times \cdots \times knd_n \mapsto \{1{:}knd_1, \ldots, n{:}knd_n\}$
$knd^n \mapsto \{1{:}knd, \ldots, n{:}knd\}$

$\mathsf{Unit} \mapsto \{\}$
$\mathsf{Bool}_{\langle lab \rangle} \mapsto \Sigma_{\langle lab \rangle}\,(\overline{\mathrm{true}}{\mapsto}\mathsf{Unit}, \overline{\mathrm{false}}{\mapsto}\mathsf{Unit})$
$con_1 \times \cdots \times con_n \mapsto \{\overline{1}{=}con_1, \cdots, \overline{n}{=}con_n\}$
$\lambda(var_1, \ldots, var_n).con \mapsto$
$\quad \lambda var{:}\Omega^n.(\{\pi_1\, var/var_1\} \cdots \{\pi_n\, var/var_n\} con)$
$(con_1, \ldots, con_n) \mapsto \{1{=}con_1, \ldots, n{=}con_n\}$

$(exp_1, \cdots, exp_n) \mapsto \{\overline{1}{=}exp_1, \cdots, \overline{n}{=}exp_n\}$
$\lambda(var{:}con){:}con'.exp \mapsto$
$\quad \pi_{\overline{1}}\,\mathsf{fix}\ var'(var{:}con){:}con'{\mapsto}exp\ \mathsf{end}$
$\qquad var' \notin \mathrm{FV}(exp)$
$\lambda(var_1{:}con_1, \ldots, var_n{:}con_n){:}con.exp \mapsto$
$\quad \lambda(var{:}con_1 \times \cdots \times con_n){:}con.$
$\qquad \{\pi_1\, var/var_1\} \cdots \{\pi_n\, var/var_n\} exp$
$\qquad var \notin \mathrm{FV}\, exp$
$\mathsf{let}\ bnd_1, \ldots, bnd_n\ \mathsf{in}\ exp\ \mathsf{end} \mapsto$
$\quad [1{=}bnd_1, \ldots, n{=}bnd_n, (n+1){=}exp].(n+1)$
$\mathsf{catch}^{con}\ exp\ \mathsf{with}\ exp' \mapsto$
$\quad \mathsf{handle}\ exp\ \mathsf{with}\ (\lambda var{:}\mathsf{Tagged}.$
$\qquad \mathsf{iftagof}\ var\ \mathsf{is}\ basis.\mathrm{fail}^*.\mathrm{tag}$
$\qquad\quad \mathsf{then}\ \lambda var{:}\mathsf{Unit}.exp'\ \mathsf{else}\ \mathsf{raise}^{con}\ var$
$\quad var \notin \mathrm{FV}(exp')$
$\mathsf{false} \mapsto \mathsf{inj}^{\mathsf{Bool}}_{\mathrm{false}}\{\}$
$\mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3 \mapsto$
$\quad \mathsf{case}^{\mathsf{Bool}}\ exp_1\ \mathsf{of}\ \lambda var{:}\mathsf{Bool}_1.exp_2, \lambda var{:}\mathsf{Bool}_2.exp_3\ \mathsf{end}$
$\qquad var \notin \mathrm{FV}(exp_2, exp3)$
$exp_1\ \mathsf{and}\ exp_2 \mapsto \mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ \mathsf{false}$

D.2 Expressions

$$\frac{}{\Gamma \vdash scon \rightsquigarrow scon : \mathrm{type}(scon)} \tag{87}$$

Rule 87: We assume a meta-level function type which gives the IL type of each constant.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : con \qquad \text{Rule 89 does not apply.}}{\Gamma \vdash longid \rightsquigarrow path : con} \tag{88}$$

Rule 88: Monomorphic variables.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : con \to con'}{\Gamma \vdash longid \rightsquigarrow \partial\,(path) : con \rightharpoonup con'} \tag{89}$$

Rule 88: Monomorphic value constructors.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \to [\mathrm{it}{:}con] \qquad \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod : sig \qquad \text{Rule 91 does not apply.}}{\Gamma \vdash longid \rightsquigarrow path(mod).\mathrm{it} : con} \tag{90}$$

Rule 90: Polymorphic variables. The module *mod* is the structure of types (and equality functions) that we "guess" to instantiate the polymorphism.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \to [\mathrm{it}{:}con \to con'] \qquad \Gamma \vdash_{\mathrm{inst}} \rightsquigarrow mod : sig}{\Gamma \vdash longid \rightsquigarrow \partial\,(path(mod).\mathrm{it}) : con \rightharpoonup con'} \tag{91}$$

Rule 91: Polymorphic value constructors.

$$\frac{\begin{array}{c}\sigma \text{ a permutation of } 1..n \\ var_1, \cdots, var_n \notin \mathrm{BV}(\Gamma) \\ lab_{\sigma(1)} < \cdots < lab_{\sigma(n)} \\ \forall i \in 1..n: \quad \Gamma \vdash expr_i \rightsquigarrow exp_i : con_i\end{array}}{\begin{array}{l}\Gamma \vdash \ \mathtt{\{}lab_1\ \mathtt{=}\ expr_1, \cdots, lab_n\ \mathtt{=}\ expr_n\mathtt{\}} \rightsquigarrow \\ \quad \mathsf{let}\ var_1{=}exp_1, \ldots, var_n{=}exp_n\ \mathsf{in} \\ \quad \{\overline{lab_{\sigma(1)}}{=}var_{\sigma(1)}, \cdots, \overline{var_{\sigma(n)}}{=}var_{\sigma(n)}\}\ \mathsf{end}\ : \\ \quad \{\overline{lab_{\sigma(1)}}{:}con_{\sigma(1)}, \cdots, \overline{lab_{\sigma(n)}}{:}con_{\sigma(n)}\}\end{array}} \tag{92}$$

Rule 92: The order in which labels appear in the record type is significant for the IL, so in the translation we normalize record types by sorting the labels with some fixed ordering $<$. Note that though the order of the records fields is given by this ordering, the components are evaluated (and side-effects occur) in the order that they are listed in the EL.

$$\frac{\begin{array}{c}\Gamma \vdash strdec \rightsquigarrow sbnds : sdecs \\ var \notin \mathrm{BV}(\Gamma) \qquad \Gamma, 1^{*}{\triangleright}var{:}[sdecs] \vdash expr \rightsquigarrow exp : con' \\ \Gamma, var{:}[sdecs] \vdash con' \equiv con : \Omega \qquad \Gamma \vdash con : \Omega\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{let}\ strdec\ \mathtt{in}\ expr\ \mathtt{end} \rightsquigarrow \\ \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end}\ : con\end{array}} \tag{93}$$

Rule 93: The declarations *strdec* are translated into the components of a structure; the "starred structure" convention is used here to make these components accessible while translating *expr*. Standard ML prohibits the type of the body from depending on abstract types defined locally—in particular, values created from a datatype cannot escape the scope of that datatype.

$$\frac{\begin{array}{c}\Gamma \vdash expr \rightsquigarrow exp : con'' \rightharpoonup con \\ \Gamma \vdash expr' \rightsquigarrow exp' : con' \\ \Gamma \vdash con' \equiv con'' : \Omega\end{array}}{\Gamma \vdash expr\ expr' \rightsquigarrow exp\ exp' : con} \tag{94}$$

Rule 94: General application.

$$\frac{\Gamma \vDash_{\text{ctx}} \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{con}' \to \mathit{con} \qquad \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}'}{\Gamma \vdash \mathit{longid}\ \mathit{expr}' \rightsquigarrow \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{95}$$

Rule 95: Application of monomorphic value constructor.

$$\frac{\Gamma \vDash_{\text{ctx}} \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{sig} \to [\text{it}{=}\mathit{con}' \to \mathit{con}] \qquad \Gamma \vDash_{\text{inst}} \rightsquigarrow \mathit{mod} : \mathit{sig} \qquad \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}'}{\Gamma \vdash \mathit{longid}\ \ \mathit{expr}' \rightsquigarrow (\mathit{path}\,(\mathit{mod}).\text{it})\ \mathit{exp}' : \mathit{con}} \tag{96}$$

Rule 95: Application of polymorphic value constructor.

$$\frac{\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \qquad \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con}' : \Omega \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega}{\Gamma \vdash \mathit{expr}\ \texttt{:}\ \mathit{ty} \rightsquigarrow \mathit{exp} : \mathit{con}} \tag{97}$$

Rule 97: Type constraints on expressions are verified, but do not appear in the translation.

$$\frac{\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \qquad \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp}' : \mathsf{Tagged} \rightharpoonup \mathit{con}' \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega \qquad \mathit{var} \notin \mathrm{BV}(\Gamma)}{\Gamma \vdash \mathit{expr}\ \texttt{handle}\ \mathit{match} \rightsquigarrow \mathsf{handle}\ \mathit{exp}\ \mathsf{with}\ \lambda(\mathit{var}{:}\mathsf{Tagged}){:}\mathit{con}.\ (\mathsf{catch}^{\mathit{con}}\ \mathit{exp}'\ \mathit{var}\ \mathsf{with}\ \mathsf{raise}^{\mathit{con}}\ \mathit{var}) : \mathit{con}} \tag{98}$$

Rule 98: The handling expression $\mathit{exp}'\ \mathit{var}$ may fail if the handler pattern does not match the exception raised, in which case we propagate the exception.

$$\frac{\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathsf{Tagged} \qquad \Gamma \vdash \mathit{con} : \Omega}{\Gamma \vdash \texttt{raise}\ \mathit{expr} \rightsquigarrow \mathsf{raise}^{\mathit{con}}\ \mathit{exp} : \mathit{con}} \tag{99}$$

Rule 99: The translation of a `raise` expression can be given any valid type con.

$$\frac{\mathit{var} \notin \mathrm{BV}(\Gamma) \qquad \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp} : \mathit{con}_1 \rightharpoonup \mathit{con}_2}{\Gamma \vdash \texttt{fn}\ \mathit{match} \rightsquigarrow \lambda(\mathit{var}{:}\mathit{con}_1){:}\mathit{con}_2.\ (\mathsf{catch}^{\mathit{con}_2}\ \mathit{exp}\ \mathit{var}\ \mathsf{with}\ \mathsf{raise}^{\mathit{con}_2}\ \mathit{basis}.\overline{\texttt{Match}}^*.\overline{\texttt{Match}}) : \mathit{con}_1 \rightharpoonup \mathit{con}_2} \tag{100}$$

Rule 100: The expression $\mathit{exp}\ \mathit{var}$ will fail if the match fails; here we turn the basis.fail*.fail exception into $\mathit{basis}.\overline{\texttt{Match}}^*.\overline{\texttt{Match}}$. The resulting function has a partial type because it can (syntactically) raise an exception.

$$\frac{\Gamma \vdash \mathit{expr}_1 \rightsquigarrow \mathit{exp}_1 : \mathit{con}_1 \qquad \Gamma \vdash \mathit{expr}_2 \rightsquigarrow \mathit{exp}_2 : \mathit{con}_2 \qquad \Gamma \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \Omega \qquad \Gamma \vDash_{\text{eq}} \mathit{con}_1 \rightsquigarrow \mathit{exp}_{\mathrm{v}}}{\Gamma \vdash \mathit{expr}_1\ \texttt{=}\ \mathit{expr}_2 \rightsquigarrow \mathit{exp}_{\mathrm{v}}\,(\mathit{exp}_1, \mathit{exp}_2) : \mathsf{Bool}} \tag{101}$$

Rule 101: Translation of equality comparison; $\mathit{exp}_{\mathrm{v}}$ is the equality function, having type $\mathit{con} \times \mathit{con} \rightharpoonup \mathsf{Bool}$.

D.3 Matches

$$\frac{\mathit{var}, \mathit{var}' \notin \mathrm{BV}(\Gamma) \qquad \Gamma \vdash \mathit{con}' : \Omega \qquad \Gamma \vdash \mathit{pat} \Leftarrow \mathit{var}' : \mathit{con}'\ \mathsf{else}\ \mathit{basis}.\text{fail}^*.\text{fail} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs} \qquad \Gamma, 1^*{\triangleright}\mathit{var}{:}[\mathit{sdecs}] \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}}{\Gamma \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr} \rightsquigarrow \lambda(\mathit{var}'{:}\mathit{con}'){:}\mathit{con}.\mathsf{let}\ \mathit{var}{=}[\mathit{sbnds}]\ \mathsf{in}\ \mathit{exp}\ \mathsf{end} : \mathit{con}' \rightharpoonup \mathit{con}} \tag{102}$$

Rule 102:

The result of translating a match is a function that may fail if the match fails.

$$\frac{\mathit{var} \notin \mathrm{BV}(\Gamma) \qquad \Gamma \vdash \mathit{mrule} \rightsquigarrow \mathit{exp} : \mathit{con}_1 \rightharpoonup \mathit{con}_2 \qquad \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp}' : \mathit{con}_1' \rightharpoonup \mathit{con}_2' \qquad \Gamma \vdash \mathit{con}_1 \rightharpoonup \mathit{con}_2 \equiv \mathit{con}_1' \rightharpoonup \mathit{con}_2' : \Omega}{\Gamma \vdash \mathit{mrule}\ \texttt{|}\ \mathit{match} \rightsquigarrow \lambda(\mathit{var}{:}\mathit{con}_1){:}\mathit{con}_2.\ \mathsf{catch}^{\mathit{con}}\ \mathit{exp}\ \mathit{var}\ \mathsf{with}\ \mathit{exp}'\ \mathit{var} : \mathit{con}' \rightharpoonup \mathit{con}} \tag{103}$$

Rule 103: The failure of pattern matching in the first clause is caught, and we try again with the next clause.

D.4 Polymorphic Instantiation

$$\frac{\mathit{decs} \vdash \mathit{con} : \Omega \qquad \langle \mathit{decs} \vDash_{\text{eq}} \mathit{con} \rightsquigarrow \mathit{exp}_{\mathrm{v}} \rangle \qquad \langle\langle \mathit{decs} \vDash_{\text{inst}} \rightsquigarrow [\mathit{sbnds}_{\mathrm{v}}] : [\mathit{sdecs}] \rangle\rangle}{\mathit{decs} \vDash_{\text{inst}} \rightsquigarrow [\mathit{lab}'{=}[\mathit{lab}{\triangleright}\mathit{var}{=}\mathit{con}\langle, \text{eq}{=}\mathit{exp}_{\mathrm{v}}\rangle]\langle\langle, \mathit{sbnds}_{\mathrm{v}}\rangle\rangle] \ [\mathit{lab}'{:}[\mathit{lab}{\triangleright}\mathit{var}{:}\Omega{=}\mathit{con}\langle, \text{eq}{:}\mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle]]\langle\langle, \mathit{sdecs}\rangle\rangle} \tag{104}$$

Rule 104 Nondeterministically choose types and the corresponding equality functions so as to match a fully-transparent signature.

D.5 Declarations

$$\frac{\begin{array}{c} var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash expr \leadsto exp : con \\ \Gamma, var{:}con \vdash pat \Leftarrow var : con \;\textsf{else}\; basis.\overline{\mathtt{Bind}}^{*}.\overline{\mathtt{Bind}} \leadsto \\ \quad sbnds : sdecs \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{val\ ()}\ pat\ \mathtt{=}\ expr \leadsto \\ \quad 1{\triangleright}var{=}exp, sbnds : \\ \quad 1{\triangleright}var{:}con, sdecs \end{array}} \qquad (105)$$

Rule 105: Monomorphic, non-recursive binding.

$$\frac{\begin{array}{l} sig = [\overline{tyvar_1}^{*}{:}[\overline{tyvar_1}{:}\Omega], \cdots, \overline{tyvar_n}^{*}{:}[\overline{tyvar_n}{:}\Omega], \\ \qquad 1^{*}{:}\Omega, \cdots, m^{*}{:}\Omega] \\ \Gamma, 1^{*}{\triangleright}var{:}sig \vdash expr \leadsto exp : con \\ \Gamma, 1^{*}{\triangleright}var{:}sig \vdash basis.\overline{\mathtt{Bind}}^{*}.\overline{\mathtt{Bind}} \\ \qquad\qquad \textsf{else}\; pat \Leftarrow exp : con \leadsto \\ \qquad\qquad lab_1{=}exp_1, \ldots, lab_n{=}exp_n : \\ \qquad\qquad lab_1{:}con_1, \ldots, lab_n{=}exp_n \\ \forall i \in 1..n : \\ \quad \Gamma, 1^{*}{\triangleright}var{:}sig \vdash exp_i \downarrow con_i \\ \quad sbnd'_i := lab_i{=}(var{:}sig){\rightarrow}[\mathrm{it}{=}exp_i] \\ \quad sdec'_i := lab_i{:}(var{:}sig){\rightarrow}[\mathrm{it}{:}con_i] \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{val\ (}\langle eq\rangle_1 tyvar_1, \cdots, \langle eq\rangle_n tyvar_n\mathtt{)}\ pat\ \mathtt{=}\ expr \leadsto \\ \quad sbnd'_1, \ldots, sbnd'_n : sdec'_1, \ldots, sdec'_n \end{array}} \qquad (106)$$

Rule 106: Polymorphic, non-recursive `val` bindings. (For space reasons, we show a simplified version of the full rule, which must take equality type variables into account; this version has the effect of only allowing polymorphism for irrefutable patterns.) Note that type inference may introduce new type variables not mentioned in the source (as in `val f = fn x => x`).

$$\frac{\begin{array}{c} \Gamma \vdash strdec_1 \leadsto sbnds_1 : sdecs_1 \\ \Gamma, sdecs_1 \vdash strdec_2 \leadsto sbnds_2 : sdecs_2 \end{array}}{\begin{array}{l} \Gamma \vdash strdec_1\ strdec_2 \leadsto \\ \quad sbnds_1{+}{+}sbnds_2 : sdecs_1{+}{+}sdecs_2 \end{array}} \qquad (107)$$

Rule 107: Sequential declarations are modelled with a syntactic append, except we must rename any labels (preserving the star convention) in $sbnds_1/sdecs_1$ appearing in $sbnds_2/sdecs_2$,

$$\frac{\forall i \in 1..n : \quad \Gamma \vdash_{\mathrm{ctx}} \overline{longstrid_i} \leadsto path_i : sig_i}{\begin{array}{l} \Gamma \vdash \textsf{open}\ longstrid_1\ \cdots\ longstrid_n \leadsto \\ \quad 1^{*}{=}path_1, \cdots, n^{*}{=}path_n : \\ \quad 1^{*}{:}sig_1, \cdots, n^{*}{=}sig_n \end{array}} \qquad (108)$$

$$\frac{\Gamma \vdash ty \leadsto con : \Omega \qquad var \notin \mathrm{BV}(\Gamma)}{\begin{array}{l} \Gamma \vdash \mathtt{exception}\ id\ \mathtt{of}\ ty \leadsto \\ \quad \overline{id}^{*}{=}[\mathrm{tag}{\triangleright}var{=}\textsf{new\_tag}[con], \qquad : \\ \qquad \overline{id}{=}\lambda(var'{:}con){:}\textsf{Tagged.tag}(var, var')] \\ \quad \overline{id}^{*}{:}[\mathrm{tag}{\triangleright}var{:}con\ \textsf{Tag}, \overline{id}{:}con{\rightarrow}\textsf{Tagged}] \end{array}} \qquad (109)$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathrm{ctx}} \overline{longid} \leadsto path.lab : con \\ \Gamma \vdash path.\mathrm{tag} : con' \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{exception}\ id\ \mathtt{=}\ longid \leadsto \\ \quad \overline{id}^{*}{=}[\mathrm{tag}{=}path.\mathrm{tag}, \overline{id}{=}path.lab] : \\ \quad \overline{id}^{*}{:}[\mathrm{tag}{:}con', \overline{id}{:}con] \end{array}} \qquad (110)$$

Rule 110: We know that *longid* corresponds to an exception constructor because of the tag component.

$$\frac{\begin{array}{c} var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash strdec_1 \leadsto sbnds_1 : sdecs_1 \\ \Gamma, 1^{*}{\triangleright}var{:}[sdecs_1] \vdash strdec' \leadsto sbnds_2 : sdecs_2 \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{local}\ strdec\ \mathtt{in}\ strdec'\ \mathtt{end} \leadsto \\ \quad 1{\triangleright}var{=}[sbnds_1], sbnds_2 : 1{\triangleright}var{:}[sbnds_2], sdecs_2 \end{array}} \qquad (111)$$

Rule 111: We create a bindings for all of the declarations, but the local bindings are segregated into a substructure inaccessible from the EL.

$$\frac{\Gamma \vdash tybind \leadsto sbnds : sdecs}{\Gamma \vdash \mathtt{type}\ tybind \leadsto sbnds : sdecs} \qquad (112)$$

D.6 Structure Expressions

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longstrid} \leadsto path : sig}{\Gamma \vdash longstrid \leadsto path : sig} \qquad (113)$$

$$\frac{\Gamma \vdash strdec \leadsto mod : sig}{\Gamma \vdash \mathtt{struct}\ strdec\ \mathtt{end} \leadsto mod : sig} \qquad (114)$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathrm{ctx}} \overline{longfunid} \leadsto path_f : (var_1{:}sig_1){\rightharpoonup}sig_2 \\ \Gamma \vdash_{\mathrm{ctx}} \overline{longstrid} \leadsto path : sig \\ \Gamma \vdash_{\mathrm{sub}} path : sig \preceq sig_1 \leadsto mod : sig' \\ \Gamma \vdash (var_1{:}sig){\rightharpoonup}sig_2 \equiv sig'{\rightharpoonup}sig'' : \textsf{Sig} \end{array}}{\Gamma \vdash longfunid(longstrid) \leadsto (path_f{:}sig'{\rightharpoonup}sig'')\ mod : sig''} \qquad (115)$$

Rule 115: We insert an explicit coercion to drop and reorder components of the argument structure (which has signature sig), in order to match the domain signature of the functor (sig_1). The signature sig_c is the most-specific (and fully transparent) signature of the coerced structure, which may expose more types (is a sub-signature of) sig_1.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} longstrid \leadsto path : sig \quad \Gamma \vdash sigexp \leadsto sig' : \mathsf{Sig} \quad \Gamma \vdash_{\mathrm{sub}} path : sig \preceq sig' \leadsto mod : sig''}{\Gamma \vdash longstrid \; : \; sigexp \leadsto mod : sig''} \tag{116}$$

Rule 116: Ascribing a signature to a structure using ":" hides components (this hiding being accomplished here via an explicit coercion), but allows the identity of the remaining type components to leak through. The rules for coercions ensure that sig'' will be fully transparent, maximizing propagation of type information.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} longstrid \leadsto path : sig \quad \Gamma \vdash sigexp \leadsto sig' : \mathsf{Sig} \quad \Gamma \vdash_{\mathrm{sub}} path : sig \preceq sig' \leadsto mod : sig''}{\Gamma \vdash longstrid \; : \; sigexp \leadsto (mod{:}sig') : sig'} \tag{117}$$

Rule 117: Ascribing a signature to a structure with :> not only hides components, but restricts information about types to that which appears in the signature.

$$\frac{var \notin \mathrm{BV}(\Gamma) \quad \Gamma \vdash strdec \leadsto sbnds : sdecs \quad \Gamma, 1^{*}{\triangleright}var{:}[sdecs] \vdash strexp \leadsto mod : sig}{\Gamma \vdash \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end} \leadsto [1{\triangleright}var{=}[sbnds], 2^{*}{=}mod] : [1{\triangleright}var{:}[sdecs], 2^{*}{:}sig]} \tag{118}$$

D.7 Pattern Compilation

$$\frac{lab \text{ fresh} \qquad \mathrm{type}(scon) = con \quad \Gamma \vdash exp : con}{\Gamma \vdash scon \Leftarrow exp : con \ \mathsf{else}\ exp' \leadsto lab{=}\mathsf{if}\ exp{=}_{con} scon\ \mathsf{then}\ \{\}\ \mathsf{else}\ \mathsf{raise}^{\mathsf{Unit}}\ exp' : lab{:}\mathsf{Unit}} \tag{119}$$

Rule 119: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{\Gamma \vdash con \equiv \{lab'_1{:}con'_1, \cdots, lab'_k{:}con'_k\} : \Omega \quad \{\overline{lab_1}, \cdots, \overline{lab_n}\} \subseteq \{lab'_1, \cdots, lab'_n\} \quad \forall i \in 1..n : \ \Gamma, lab{\triangleright}var{:}con \vdash pat_i \Leftarrow \pi_{\overline{lab_i}}\, exp : con_i \ \mathsf{else}\ exp' \leadsto sbnds_i : sdecs_i}{\Gamma \vdash \{lab_1 = pat_1, \ldots, lab_n = pat_n \langle, \ldots\rangle\} \ \mathsf{else}\ exp' \Leftarrow exp : con \leadsto sbnds_1, \ldots, sbnds_n : sdecs_1, \ldots, sdecs_n} \tag{120}$$

Rule 120: Pattern match against a record of patterns. Because we disallow repeated variables in patterns, the syntactic concatenation of structure here is well-formed.