A Type-Theoretic Interpretation of Standard ML*

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

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

1 Introduction

It has been nearly twenty years since Robin Milner introduced ML as the metalanguage of the LCF interactive theorem prover [5]. His elegant use of abstract types to ensure validity of machine-generated proofs, combined with his innovative and flexible polymorphic type discipline, and supported by his rigorous proof of soundness for the language, inspired a large body of research into the type structure of programming languages.[1] As a design tool type theory gives substance to informal ideas such as "orthogonality" and "safety" and provides a framework for evaluating and comparing languages. As an implementation tool type theory provides a framework for structuring compilers and supports the use of efficient data representations even in the presence of polymorphism [28, 27].

Milner's work on ML culminated in his ambitious proposal for Standard ML [17] that 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. These challenges were met in *The Definition of Standard ML* (hereafter, *The Definition*), which provided a precise definition of the static and dynamic semantics in a uniform relational framework. A key difficulty in the formulation of the static semantics of Standard ML is to manage the propagation of type information in a program so as to support data abstraction while avoiding excessive notational burdens on the programmer. This is achieved in *The Definition* through the use of "generative stamps". Roughly speaking, each type is assigned a unique "stamp" that serves as proxy for the underlying representation of that type. This ensures that two abstract types with the same representation are distinguished from one another, and facilitates the use of type inference-based techniques in the semantics of modules. The stamp formalism does not inhibit the dynamic semantics of the language because types are erased prior to execution. Consequently, no management of stamps is required at run-time.

Much recent research on both the metatheory and implementation of programming languages is based on an *explicitly-typed* interpretation in which the dynamic semantics is defined on typed, rather than type-erased, programs [28, 2]. From a semantic viewpoint programs are seen as intrinsically typed, and hence fundamental relations such as operational equivalence are defined so as to limit the set of observations to those that make type sense. From an implementation viewpoint types are used to determine the behavior of primitive operations (such as array subscripting) and to perform storage management at run-time. The

*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 comprehensive survey and references to the literature.

untyped interpretation is a special case of the typed interpretation in which we consider only one (typically recursive) type. Thus there is no loss of generality in considering the explicitly-typed case.

While there has been considerable progress in developing a type-theoretic account of programming languages, a complete treatment of fully-featured languages such as Standard ML has thus far not been achieved. One obstacle is scale: Standard ML has a rich collection of mechanisms that must be accounted for in any formal treatment. More significantly, Standard ML presents a number of challenges to a type-theoretic account, principally the module system. For example, it is not immediately clear how to extend the generative stamps formalism of *The Definition* to an explicitly-typed setting. The main difficulty is that in a typed framework the underlying representation of an abstract type must be exposed at run-time. Consequently, an explicit association between stamps and their implementation types must be maintained in a typed semantics. Other aspects of Standard ML, including recursive datatype declarations, pattern matching, polymorphic equality, and "generative" functors, also present significant challenges for a type-theoretic interpretation.

In this paper we outline an interpretation of Standard ML in a typed framework. The interpretation takes the form of a translation from 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*, or *EL*. The external language considered here is the 1997 dialect of Standard ML, as described in the revised *Definition* [18]. 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* [13], 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 static semantic objects. 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. 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 an abstract type with its underlying representation.

The interpretation may be viewed as an alternative to *The Definition* in which the "static semantic objects" have been formalized as expressions of a typed λ-calculus and in which the elaboration relation has been generalized to a translation into the internal language. From this point of view the internal language plays a role analogous to Scott's LAMBDA language for denotational semantics [25]. The meaning of a Standard ML program is defined by interpretation into the internal language, which is given meaning by some other means. In our setting the semantics of the internal language is given by a sound operational semantics, but we conjecture that it would also be feasible to give it a domain-theoretic interpretation as in denotational semantics.

The interpretation may also be viewed as a declarative specification of the elaboration rules for type-based compilers for Standard ML such as TIL [28] and SML/NJ [26]. The front-end of the TIL compiler is a "determinization" of elaboration rules described below, using standard methods such as unification to defer non-deterministic choices until the context resolves any ambiguity. Preliminary results indicate that basing a compiler on a typed interpretation has numerous advantages, both in terms of expressive power (resolving a long-standing difficulty with the compilation of functors in Standard ML) and efficiency (leading to significant improvements in space and time requirements).

The internal language is intended to capture the fundamental constructs shared by many programming languages. We conjecture that languages such as Caml, Haskell, and Scheme could be interpreted into an internal language substantially similar to the one we give here. For example, we may translate Scheme expressions into internal language expressions of a fixed recursive sum type. Correspondingly, the primitive operations dispatch on the form values, much as actual Scheme implementations analyze tag bits at run-time. The interpretation framework neatly handles Scheme's decision to leave the evaluation order of function arguments unspecified — the translation rules can rely on the indeterminacy of the relational framework to "choose" an evaluation order at each application expression. Direct approaches to the semantics of Scheme have great difficulty accounting for this aspect of the language.

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* : 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* : Sig	*sig* is well-formed
decs $\vdash$ *sdecs* $\leq$ *sdecs'*	component-wise subtyping
decs $\vdash$ *sig* $\leq$ *sig'* : Sig	signature subtyping
decs $\vdash$ *sdecs* $\equiv$ *sdecs'*	component-wise equivalence
decs $\vdash$ *sig* $\equiv$ *sig'* : 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

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, 15] translucent sum formalism and Leroy's manifest type system [13].

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 [11]. 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.

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 [20] 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 [31, 12] 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.

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

Figure 2: Constructors and Kinds

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 [22], 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.

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.

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

$exp ::=$	$scon$	constants	|	$\mathsf{inj}^{con}_{lab}\ exp$	injection into sum
|	var	variables	|	$\mathsf{proj}^{con}_{lab}\ exp$	sum projection
|	loc	memory locations	|	$\mathsf{case}^{con}\ exp\ \mathsf{of}\ exp_1, \ldots\ \mathsf{end}$	sum case analysis
|	tag	exception tags	|	$\mathsf{new_tag}[con]$	extend type Tagged
|	$\mathsf{fix}\ fbnds\ \mathsf{end}$	recursive functions	|	$\mathsf{tag}(exp, exp)$	injection into Tagged
|	$exp\ exp'$	application	|	$\mathsf{iftagof}\ exp\ \mathsf{is}\ exp'$	tag analysis
|	$\{rbnds\}$	record expression		$\mathsf{then}\ exp''\ \mathsf{else}\ exp'''$	
|	$\pi_{lab}\ exp$	record projection	|	$exp_1 =_{\mathsf{Int}} exp_2, \ldots$	base equalities
|	$\mathsf{handle}\ exp\ \mathsf{with}\ exp'$	handle exception	|	$mod.lab$	module projection
|	$\mathsf{raise}^{con}\ exp$	raise exception			
|	$\mathsf{ref}^{con}\ exp$	new ref cell	$rbnds ::=$	$\cdot$	empty
|	$\mathsf{get}\ exp$	dereference	|	$rbnds, rbnd$	sequence
|	$\mathsf{set}\ (exp, exp')$	assignment	$rbnd ::=$	$lab = exp$	record field binding
|	$\mathsf{roll}^{con}\ exp$	coerce to μ type			
|	$\mathsf{unroll}^{con}\ exp$	coerce from μ type	$fbnds ::=$	$\cdot$	empty
|	$\partial\ exp$	coerce total to partial	|	$fbnds, fbnd$	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

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, 13]. 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 [19] that affords fine-grained control over the "degree" of abstractness of a type. They may also be seen as a variant of the "dependent sum" type [16], 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

$mod ::=$	var	module variable	$sig ::=$	$[sdecs]$	structure signature
$\mid$	$[sbnds]$	structure	$\mid$	$(var{:}sig){\rightharpoonup}sig'$	partial functor signature
$\mid$	$\lambda var{:}sig.mod$	functor	$\mid$	$(var{:}sig){\rightarrow}sig'$	total functor signature
$\mid$	$mod\ mod'$	functor application			
$\mid$	$mod.lab$	structure projection	$sdecs ::=$	$\cdot$	structure field dec.
$\mid$	$mod{:}sig$	signature ascription	$\mid$	$sdecs, sdec$	
			$sdec ::=$	$lab{:}dec$	
$sbnds ::=$	$\cdot$	structure field bindings			
$\mid$	$sbnds, sbnd$		$decs ::=$	$\cdot$	declaration lists
$sbnd ::=$	$lab{:}bnd$		$\mid$	$decs, dec$	
			$dec ::=$	$var{:}con$	expression variable dec.
$bnd ::=$	$var{=}con$	constructor binding	$\mid$	$var{:}sig$	module variable dec.
$\mid$	$var{=}exp$	expression binding	$\mid$	$var{:}knd$	opaque type dec.
$\mid$	$var{=}mod$	module binding	$\mid$	$var{:}knd{=}con$	transparent type dec.
			$\mid$	$loc{:}con$	typed locations
			$\mid$	$tag{:}con$	typed exception tag

Figure 4: Modules and Signatures

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 and 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. In the general translucent sum calculus this may not be possible for all arguments; however we have made syntactic restrictions in our IL so that the dependency can always be eliminated.

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

Figure 5: Internal Language Values

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:

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

Thus 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 ensures that abstraction boundaries are respected. Specifically, 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 [23] and Wright and Felleisen [32], 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

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

Figure 6: Evaluation Contexts

- 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.

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

$(\Delta, \sigma, exp_{\mathrm{v}})$	normal termination
$(\Delta, \sigma, \mathsf{raise}^{con}\ exp_{\mathrm{v}})$	uncaught exception

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", written [] (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.

We have given a high-level operational semantics in that types are propagated, but never normalized or examined at run-time. To describe primitives which do intensional type analysis [10] we could refine the semantics to perform normalization computations at the constructor level as well.

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 [32], 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 [29].

Fix a base type *ans* of *answers* to which a complete, closed program might evaluate.[2] We 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.

[2] 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.

Proposition 4 (Canonical Forms)

1. *Assume* $\Delta \vdash exp_v' : con'$ *where* exp_v' *is closed.*

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

2. *Assume* $\Delta \vdash mod_v' : sig'$ *where* mod_v' *is closed.*

If sig' *is of the form...*	*then* mod_v' *is of the form...*
$[sdecs]$	$[sbnds_v]$
$var{:}sig \rightharpoonup sig'$	$\lambda var{:}sig.mod$
$var{:}sig \to sig'$	$\lambda var{:}sig.mod$

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

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

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:

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 the type together with the associated equality operation. Datatypes that admit equality are equipped with an equality operation.

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

Figure 7: Judgment forms for the Elaboration

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 [14]. 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 rules use a number of "derived forms" in the internal language; these are shown in Appendix D.1. Specifics of the elaboration process are 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[3] 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. Thus, for example, we have the translation

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

but, in contrast, when L is open we have

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

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). Datatype constructors (functions with total types) are translated to user-level functions (with partial types) when used as values. Application translates to internal-language application, with a check to ensure that 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 order while preserving the order of evaluation. 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

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

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

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

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

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 contains 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; this 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, 24]). 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. 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[4] to the functor

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

with signature

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

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 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

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

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

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

"generativity" of `datatype` declarations in Standard ML; the declared type is "new" in the sense that it is represented by a path that is, by α-conversion, distinct from all previous 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 $\mathit{sig}_{\texttt{list}}$ given in Figure 8. Viewed as a declaration, this phrase elaborates into the sealed structure $\mathit{mod}_{\texttt{list}}{:}\mathit{sig}_{\texttt{list}}$, where $\mathit{mod}_{\texttt{list}}$ is given in Figure 9.

The signature $\mathit{sig}_{\texttt{list}}$ 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 `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 $\mathit{mod}_{\texttt{list}}$ implements the signature $\mathit{sig}_{\texttt{list}}$. 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 provide an equal operation for "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.[5]

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 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

[5] It is a questionable feature of Standard ML that such types admit equality such that the equality operation they admit is not definable in SML itself.

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. 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 [21]), 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.

We note that 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 cleanly 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. 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 [30] and to the treatment of equality proposed by Gunter, Gunter and MacQueen [6].

The judgment $\Gamma \vdash_{\bar{\mathrm{eq}}} con \rightsquigarrow exp_{\mathrm{v}}$ 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
```

augments the structure containing the type component (equal to `Int`) with an equality function (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 [14] 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 make a prepass introduces bindings for non-trivial module expression [14]. The practical effect of the restriction to named form is that the result of every functor 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.[6] 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.

[6] 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.

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. We could then 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:

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

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

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

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

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

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

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 typed λ-calculus. A fully detailed account of the interpretation appears in a companion technical report [11]. 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 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 relatively 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 state soundness in the framework of *The Definition* requires that the dynamic semantics be instrumented with such error transitions, which would significantly increase the number of rules required. Finally, we note that the internal language does not rely on any external 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 its complete form. 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 at non-base types 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

since the association between an abstract type and its representation must be made explicit in the dynamic semantics, and this is 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 *The Definition* by 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.

The type-theoretic interpretation has both advantages and disadvantages as an alternative 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 psychological question of whether the kind of translation we give here can serve as a useful reference for programmers. 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 correspond to different interpretations of the language.

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.

[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, *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 LNCS 78. Springer-Verlag, 1979.

[6] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. An abstract interpretation for ML equality kinds. *LNCS 526*, pages 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 Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, 1995.

[11] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, 1997.

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

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

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

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

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

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

[18] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

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

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

[22] 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, 18th International Symposium*, volume 711 of *LNCS*, pages 122–141. Springer-Verlag, Berlin, 1993.

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

[24] 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.

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

[26] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, June 1997.

[27] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, 1995.

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

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

[30] 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.

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

[32] 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

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{scon} \\
& | & \mathit{longid} \\
& | & \{\mathit{lab}_1 \texttt{ = } \mathit{expr}_1, \cdots, \mathit{lab}_n \texttt{ = } \mathit{expr}_n\} \\
& | & \texttt{let } \mathit{strdec} \texttt{ in } \mathit{expr} \texttt{ end} \\
& | & \mathit{expr}\ \mathit{expr}' \\
& | & \mathit{expr} \texttt{ : } \mathit{ty} \\
& | & \mathit{expr} \texttt{ handle } \mathit{match} \\
& | & \texttt{raise } \mathit{expr} \\
& | & \texttt{fn } \mathit{match} \\
& | & \mathit{expr}_1 \texttt{ = } \mathit{expr}_2 \\
\\
\mathit{mrule} & ::= & \mathit{pat} \texttt{ => } \mathit{expr} \\
\mathit{match} & ::= & \mathit{mrule} \\
& | & \mathit{mrule} \texttt{ | } \mathit{match} \\
\\
\mathit{strdec} & ::= & \cdot \\
& | & \texttt{val } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{pat} \texttt{ = } \mathit{exp} \\
& | & \texttt{val } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \texttt{ rec } \mathit{pat} \texttt{ = } \mathit{exp} \\
& | & \mathit{strdec}_1\ \mathit{strdec}_2 \\
& | & \texttt{open } \mathit{longid}_1\ \cdots\ \mathit{longid}_n \\
& | & \texttt{exception } \mathit{id} \\
& | & \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \\
& | & \texttt{exception } \mathit{id} \texttt{ = } \mathit{longid} \\
& | & \texttt{local } \mathit{strdec}_1 \texttt{ in } \mathit{strdec}_2 \texttt{ end} \\
& | & \texttt{type } \mathit{tybind} \\
& | & \texttt{datatype } \mathit{datbind} \\
& | & \texttt{datatype } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{tycon} \texttt{ =} \\
& & \quad \texttt{datatype } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{longtycon} \\
& | & \texttt{structure } \mathit{strbind} \\
& | & \texttt{functor } \mathit{funbind} \\
\\
\mathit{tybind} & ::= & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon} \texttt{ = } \mathit{ty}\ \langle \langle \texttt{and } \mathit{tybind} \rangle \rangle \\
\mathit{datbind} & ::= & (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{tycon} \texttt{ = } \mathit{conbind} \\
& & \langle \texttt{and } \mathit{datbind} \rangle \\
\mathit{conbind} & ::= & \mathit{id}\ \langle \texttt{of } \mathit{ty} \rangle\ \langle \langle \texttt{| } \mathit{conbind} \rangle \rangle \\
\\
\mathit{strexp} & ::= & \mathit{longstrid} \\
& | & \texttt{struct } \mathit{strdec} \texttt{ end} \\
& | & \mathit{longfunid}\ (\mathit{longstrid}) \\
& | & \mathit{longstrid} \texttt{ : } \mathit{sigexp} \\
& | & \mathit{longstrid} \texttt{ :> } \mathit{sigexp} \\
& | & \texttt{let } \mathit{strdec} \texttt{ in } \mathit{strexp} \texttt{ end}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{spec} & ::= & \cdot \\
& | & \texttt{val } \mathit{id} \texttt{ : } \mathit{ty} \\
& | & \texttt{type } \mathit{typdesc} \\
& | & \texttt{eqtype } \mathit{etypdesc} \\
& | & \texttt{datatype } \mathit{datbind} \\
& | & \texttt{datatype } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{tycon}' \texttt{ =} \\
& & \quad \texttt{datatype } (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{longtycon} \\
& | & \texttt{exception } \mathit{id} \\
& | & \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \\
& | & \texttt{structure } \mathit{strid} \texttt{ : } \mathit{sigexp} \\
& | & \texttt{functor } \mathit{funid}\ (\mathit{strid} \texttt{ : } \mathit{sigexp}) \texttt{ : } \mathit{sigexp}' \\
& | & \texttt{include } \mathit{sigexp} \\
& | & \mathit{spec}_1\ \mathit{spec}_2 \\
& | & \mathit{spec} \texttt{ sharing type } \mathit{longid}_1 \texttt{ = } \mathit{longid}_2 \\
\\
\mathit{typdesc} & ::= & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon} \\
& & \quad \langle \langle \texttt{and } \mathit{typdesc} \rangle \rangle \\
& | & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon} \texttt{ = } \mathit{ty} \\
& & \quad \langle \langle \texttt{and } \mathit{typdesc} \rangle \rangle \\
\\
\mathit{etypdesc} & ::= & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon} \\
& & \quad \langle \langle \texttt{and } \mathit{etypdesc} \rangle \rangle \\
\\
\mathit{sigexp} & ::= & \texttt{sig } \mathit{spec} \texttt{ end} \\
& | & \mathit{sigexp} \texttt{ where type} \\
& & \quad \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{longtycon} \texttt{ = } \mathit{ty} \\
\\
\mathit{pat} & ::= & \mathit{scon} \\
& | & \mathit{longid} \\
& | & \texttt{_} \\
& | & \mathit{pat} \texttt{ : } \mathit{ty} \\
& | & \mathit{longid}\ \mathit{pat} \\
& | & \{\mathit{lab}_1 \texttt{ = } \mathit{pat}_1, \cdots, \mathit{lab}_n \texttt{ = } \mathit{pat}_n \langle, \ldots \rangle\} \\
& | & \mathit{pat}_1 \texttt{ as } \mathit{pat}_2 \\
& | & \texttt{ref } \mathit{pat} \\
\\
\mathit{ty} & ::= & \mathit{base} \\
& | & \mathit{tyvar} \\
& | & \{\mathit{lab}_1 \texttt{ : } \mathit{expr}_1, \cdots, \mathit{lab}_n \texttt{ : } \mathit{expr}_n\} \\
& | & \langle (\mathit{ty}_1, \cdots, \mathit{ty}_n) \rangle\ \mathit{longtycon} \\
& | & \mathit{ty} \texttt{ -> } \mathit{ty}' \\
\\
\mathit{strbind} & ::= & \mathit{strid} \texttt{ = } \mathit{strexp}\ \langle \texttt{and } \mathit{strbind} \rangle \\
\mathit{funbind} & ::= & \mathit{funid}\ (\mathit{strid} \texttt{ : } \mathit{sigexp}) \texttt{ = } \mathit{strexp} \\
& & \quad \langle \texttt{and } \mathit{funbind} \rangle
\end{array}
$$

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. The most significant difference is the restricted form we

allow for structure expressions (*strexp*); Standard ML programs can always be put into this form by a simple prepass. We also extend the grammar to allow for module definitions local to an expression, and for functor specifications in signatures. We also 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

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

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

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \rightharpoonup con_1' \equiv con_2 \rightharpoonup con_2' : \Omega} \tag{3}$$

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \to con_1' \equiv con_2 \to con_2' : \Omega} \tag{4}$$

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

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

$$\frac{\begin{array}{c} lab_1, \ldots, lab_n \text{ distinct} \\ \forall i \in 1..n : \quad decs \vdash con_i \equiv con_i' : \Omega \end{array}}{\begin{array}{r} decs \vdash \{lab_1{:}con_1, \cdots, lab_n{:}con_n\} \equiv \\ \{lab_1{:}con_1', \cdots, lab_n{:}con_n'\} : \Omega \end{array}} \tag{7}$$

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

$$\frac{\begin{array}{c} \langle i \in 1..n \rangle \\ \forall i \in 1..n : \quad decs \vdash con_i \equiv con_i' : \Omega \end{array}}{\begin{array}{r} decs \vdash \Sigma_{\langle lab_i \rangle}\, (lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) \equiv \\ \Sigma_{\langle lab_i \rangle}\, (lab_1 \mapsto con_1', \ldots, lab_n \mapsto con_n') : \Omega \end{array}} \tag{9}$$

$$\frac{decs \vdash con_1 \equiv con_2 : knd'{\Rightarrow}knd \qquad decs \vdash con_1' \equiv con_2' : knd'}{decs \vdash con_1\ con_2 \equiv con_1'\ con_2' : knd} \tag{10}$$

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

$$\frac{decs \vdash con : knd}{decs \vdash con \equiv con : knd} \tag{12}$$

$$\frac{decs \vdash con' \equiv con : knd}{decs \vdash con \equiv con' : knd} \tag{13}$$

$$\frac{decs \vdash con \equiv con' : knd \qquad decs \vdash con' \equiv con'' : knd}{decs \vdash con \equiv con'' : knd} \tag{14}$$

B.2 Well-formed Expressions

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', var{:}con, decs''}{decs \vdash var : con} \tag{15}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', loc{:}con, decs''}{decs \vdash loc : con} \tag{16}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', tag{:}con, decs''}{decs \vdash tag : con} \tag{17}$$

$$\frac{decs \vdash exp : con' \rightharpoonup con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \tag{18}$$

$$\frac{decs \vdash exp : con' \to con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \tag{19}$$

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

$$\frac{var' \notin \mathrm{FV}\, exp \qquad decs, var{:}con \vdash exp \downarrow con'}{decs \vdash \mathsf{fix}\ var'(var{:}con){:}con' \mapsto exp\ \mathsf{end} : \{\overline{1}{:}con \to con'\}} \tag{21}$$

$$\frac{lab_1, \cdots, lab_n \text{ distinct} \qquad \forall i \in 1..n : \quad decs \vdash exp_i : con_i}{decs \vdash \{lab_1{=}exp_1, \cdots, lab_n{=}exp_n\} : \{lab_1{:}con_1, \cdots, lab_n{:}con_n\}} \tag{22}$$

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

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

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

$$\frac{decs \vdash con : \Omega}{decs \vdash \mathsf{new_tag}[con] : con\ \mathsf{Tag}} \tag{26}$$

$$\frac{decs \vdash exp : con}{decs \vdash \mathsf{ref}^{con}\ exp : con\ \mathsf{Ref}} \tag{27}$$

$$\frac{decs \vdash exp : con\ \mathsf{Ref}}{decs \vdash \mathsf{get}\ exp : con} \tag{28}$$

$$\frac{decs \vdash exp : con\ \mathsf{Ref} \qquad decs \vdash exp' : con}{decs \vdash \mathsf{set}\ (exp, exp') : \mathsf{Unit}} \tag{29}$$

$$\frac{decs \vdash con \equiv (\pi_{lab}\ (\mu\ con'))\ \langle con'' \rangle : \Omega \qquad decs \vdash exp : (\pi_{lab}\ (con'\ (\mu\ con')))\ \langle con'' \rangle}{decs \vdash \mathsf{roll}^{con}\ exp : con} \tag{30}$$

$$\frac{decs \vdash con \equiv (\pi_{lab}\ (\mu\ con'))\ \langle con'' \rangle : \Omega \qquad decs \vdash exp : con}{decs \vdash \mathsf{unroll}^{con}\ exp : (\pi_{lab}\ (con'\ (\mu\ con')))\ \langle con'' \rangle} \tag{31}$$

$$\frac{decs \vdash exp : con \to con'}{decs \vdash \partial\ exp : con \rightharpoonup con'} \tag{32}$$

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

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

$$\frac{n \geq 1 \qquad con = \Sigma(lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) \qquad decs \vdash exp : con \qquad \forall i \in 1..n : \ decs \vdash exp_i : \Sigma_{lab_i}\ (lab_1 \mapsto con_1, \ldots, lab_n \mapsto con_n) \rightharpoonup con'}{decs \vdash \mathsf{case}^{con}\ exp\ \mathsf{of}\ exp_1, \ldots, exp_n\ \mathsf{end} : con'} \tag{35}$$

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

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

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

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

B.3 Signature Subtyping

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

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

$$\frac{decs \vdash sig \leq sig' : \mathsf{Sig} \qquad decs, var{:}sig \vdash sdecs \ \leq \ sdecs'}{decs \vdash lab \triangleright var{:}sig, sdecs \ \leq \ lab \triangleright var{:}sig', sdecs'} \tag{42}$$

$$\frac{decs \vdash lab{:}dec \ \equiv \ lab{:}dec' \qquad decs, dec \vdash sdecs \ \leq \ sdecs'}{decs \vdash lab \triangleright dec, sdecs \ \leq \ lab \triangleright dec', sdecs'} \tag{43}$$

$$\frac{decs \vdash sdecs \leq sdecs'}{decs \vdash [sdecs] \leq [sdecs'] : \mathsf{Sig}} \quad (44)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash var{:}sig_1 \rightharpoonup sig_1' \leq var{:}sig_2 \rightharpoonup sig_2' : \mathsf{Sig}} \quad (45)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash var{:}sig_1 \to sig_1' \leq var{:}sig_2 \to sig_2' : \mathsf{Sig}} \quad (46)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash var{:}sig_1 \to sig_1' \leq var{:}sig_2 \to sig_2' : \mathsf{Sig}} \quad (47)$$

B.4 Well-formed Modules

$$\frac{}{decs \vdash \cdot \,:\, \cdot} \quad (48)$$

$$\frac{decs \vdash bnd \,:\, dec \qquad decs, dec \vdash sbnds \,:\, sdecs}{decs \vdash lab \triangleright bnd, sbnds \,:\, lab \triangleright dec, sdecs} \quad (49)$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', var{:}sig, decs''}{decs \vdash var : sig} \quad (50)$$

$$\frac{decs \vdash sbnds \,:\, sdecs}{decs \vdash [sbnds] : [sdecs]} \quad (51)$$

$$\frac{decs, var{:}sig \vdash mod : sig'}{decs \vdash \lambda var{:}sig.mod : var{:}sig \rightharpoonup sig'} \quad (52)$$

$$\frac{decs, var{:}sig \vdash mod \downarrow sig'}{decs \vdash \lambda var{:}sig.mod : var{:}sig \to sig'} \quad (53)$$

$$\frac{decs \vdash mod : sig' \rightharpoonup sig \qquad decs \vdash mod' : sig'}{decs \vdash mod\ mod' : sig} \quad (54)$$

$$\frac{decs \vdash mod : [sdecs, lab{:}sig, sdecs'] \qquad \mathrm{BV}(sdecs) \cap \mathrm{FV}(sig) = \emptyset}{decs \vdash mod.lab : sig} \quad (55)$$

$$\frac{decs \vdash mod : sig}{decs \vdash mod{:}sig : sig} \quad (56)$$

$$\frac{decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}knd, sdecs']}{decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}knd{=}mod_{\mathrm{v}}.lab, sdecs']} \quad (57)$$

$$\frac{decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}sig, sdecs'] \qquad decs \vdash mod_{\mathrm{v}}.lab : sig'}{decs \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright var{:}sig', sdecs']} \quad (58)$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash sig \leq sig' : \mathsf{Sig}}{decs \vdash mod : sig'} \quad (59)$$

B.5 Valuability Judgments

$$\frac{decs \vdash exp : con \qquad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \quad (60)$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \quad (61)$$

$$\frac{}{decs \vdash exp_{\mathrm{v}} \downarrow} \quad (62)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \quad (63)$$

$$\frac{decs \vdash exp_1 \downarrow con' \to con \qquad decs \vdash exp_2 \downarrow}{decs \vdash exp_1\ exp_2 \downarrow} \quad (64)$$

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

$$\frac{}{decs \vdash mod_{\mathrm{v}} \downarrow} \quad (66)$$

$$\frac{decs \vdash mod \downarrow sig' \to sig \qquad decs \vdash mod' \downarrow}{decs \vdash mod\ mod' \downarrow} \quad (67)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \quad (68)$$

C IL Dynamic Semantics

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

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

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

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

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

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

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

$$(\Delta, \sigma, E[\mathsf{new_tag}[con]]) \hookrightarrow (\Delta[tag{:}con\ \mathsf{Tag}], \sigma, E[tag]) \quad \text{if } tag \notin \mathrm{BV}(\Delta) \tag{76}$$

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

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

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

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

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

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

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

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

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

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

D Elaboration (excerpts)

D.1 Derived forms

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

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

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

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.

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

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.

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

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

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

Rule 91: Polymorphic value constructors.

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

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.

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

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.

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : con''{\rightharpoonup}con \quad \Gamma \vdash expr' \rightsquigarrow exp' : con' \quad \Gamma \vdash con' \equiv con'' : \Omega}{\Gamma \vdash expr\ \ expr' \rightsquigarrow exp\ exp' : con} \tag{94}$$

Rule 94: General application.

$$\frac{\Gamma \vdash_{\overline{\text{ctx}}} longid \rightsquigarrow path : con' \rightarrow con \quad \Gamma \vdash expr' \rightsquigarrow exp' : con'}{\Gamma \vdash longid\ \ expr' \rightsquigarrow exp\ exp' : con} \tag{95}$$

Rule 95: Application of monomorphic value constructor.

$$\frac{\Gamma \vdash_{\overline{\text{ctx}}} longid \rightsquigarrow path : sig \rightarrow [\text{it}:con' \rightarrow con] \quad \Gamma \vdash_{\overline{\text{inst}}} \rightsquigarrow mod : sig \quad \Gamma \vdash expr' \rightsquigarrow exp' : con'}{\Gamma \vdash longid\ \ expr' \rightsquigarrow (path\,(mod).\text{it})\ exp' : con} \tag{96}$$

Rule 96: Application of polymorphic value constructor.

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

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

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

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

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

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

$$\frac{var \notin \mathrm{BV}(\Gamma) \quad \Gamma \vdash match \rightsquigarrow exp : con_1 \rightharpoonup con_2}{\begin{array}{l}\Gamma \vdash \texttt{fn}\ match \rightsquigarrow \\ \quad \lambda(var{:}con_1){:}con_2. \qquad : \\ \quad (\mathsf{catch}^{con_2}\ exp\ var \\ \quad\ \mathsf{with}\ \mathsf{raise}^{con_2}\ basis.\overline{\texttt{Match}}^*.\overline{\texttt{Match}}) \\ \quad con_1 \rightharpoonup con_2\end{array}} \tag{100}$$

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

$$\frac{\Gamma \vdash expr_1 \rightsquigarrow exp_1 : con_1 \quad \Gamma \vdash expr_2 \rightsquigarrow exp_2 : con_2 \quad \Gamma \vdash con_1 \equiv con_2 : \Omega \quad \Gamma \vdash_{\overline{\text{eq}}} con_1 \rightsquigarrow exp_{\text{v}}}{\Gamma \vdash expr_1\ \texttt{=}\ expr_2 \rightsquigarrow exp_{\text{v}}\ (exp_1, exp_2) : \mathsf{Bool}} \tag{101}$$

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

D.3 Matches

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

Rule 102:

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

$$\frac{var \notin \mathrm{BV}(\Gamma) \quad \Gamma \vdash mrule \rightsquigarrow exp : con_1 \rightharpoonup con_2 \quad \Gamma \vdash match \rightsquigarrow exp' : con_1' \rightharpoonup con_2' \quad \Gamma \vdash con_1 \rightharpoonup con_2 \equiv con_1' \rightharpoonup con_2' : \Omega}{\begin{array}{l}\Gamma \vdash mrule\ \texttt{|}\ match \rightsquigarrow \\ \quad \lambda(var{:}con_1){:}con_2. \qquad : \\ \quad \mathsf{catch}^{con}\ exp\ var\ \mathsf{with}\ exp'\ var \\ \quad con' \rightharpoonup con\end{array}} \tag{103}$$

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

$$\frac{decs \vdash con : \Omega \quad \langle decs \vdash_{\overline{\text{eq}}} con \rightsquigarrow exp_{\text{v}} \rangle \quad \langle\langle decs \vdash_{\overline{\text{inst}}} \rightsquigarrow [sbnds_{\text{v}}] : [sdecs] \rangle\rangle}{\begin{array}{l}decs \vdash_{\overline{\text{inst}}} \rightsquigarrow \\ \quad [lab' {=} [lab \triangleright var {=} con\langle, \text{eq}{=}exp_{\text{v}}\rangle]\langle\langle, sbnds_{\text{v}}\rangle\rangle] \\ \quad [lab' {:} [lab \triangleright var {:} \Omega {=} con\langle, \text{eq}{:}var \times var \rightharpoonup \mathsf{Bool}\rangle]]\langle\langle, sdecs\rangle\rangle\end{array}} \tag{104}$$

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

D.5 Declarations

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

Rule 105: Monomorphic, non-recursive binding.

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

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`).

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

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 $sbnds_2/sdecs_2$,

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

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

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

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

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

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

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

D.6 Structure Expressions

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

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

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

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' is the most-specific (and fully transparent) signature of the coerced structure, which may expose more types (is a sub-signature of) sig_1.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} longstrid \rightsquigarrow path : sig \\ \Gamma \vdash sigexp \rightsquigarrow sig' : \mathsf{Sig} \\ \Gamma \vdash_{\text{sub}} path : sig \preceq sig' \rightsquigarrow mod : sig''\end{array}}{\Gamma \vdash longstrid \ : \ sigexp \rightsquigarrow mod : sig''} \quad (116)$$

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.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} longstrid \rightsquigarrow path : sig \\ \Gamma \vdash sigexp \rightsquigarrow sig' : \mathsf{Sig} \\ \Gamma \vdash_{\text{sub}} path : sig \preceq sig' \rightsquigarrow mod : sig''\end{array}}{\Gamma \vdash longstrid \ : \ sigexp \rightsquigarrow (mod{:}sig') : sig'} \quad (117)$$

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.

$$\frac{\begin{array}{c}var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash strdec \rightsquigarrow sbnds : sdecs \\ \Gamma, 1^* \triangleright var{:}[sdecs] \vdash strexp \rightsquigarrow mod : sig\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ strdec\ \mathtt{in}\ strexp\ \mathtt{end} \rightsquigarrow \\ \quad [1 \triangleright var{=}[sbnds], 2^*{=}mod] : \\ \quad [1 \triangleright var{:}[sdecs], 2^*{:}sig]\end{array}} \quad (118)$$

D.7 Pattern Compilation

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

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

$$\frac{\begin{array}{c}\Gamma \vdash con \equiv \{lab'_1{:}con'_1, \cdots, lab'_k{:}con'_k\} : \Omega \\ \{\overline{lab}_1, \cdots, \overline{lab}_n\} \subseteq \{lab'_1, \cdots, lab'_k\} \\ \forall i \in 1..n : \\ \quad \Gamma, lab \triangleright var{:}con \vdash pat_i \\ \qquad \Leftarrow \pi_{\overline{lab}_i}\ exp : con_i \ \mathsf{else}\ exp' \rightsquigarrow sbnds_i : sdecs_i\end{array}}{\begin{array}{l}\Gamma \vdash \{lab_1 = pat_1, \ldots, lab_n = pat_n \langle, \ldots \rangle\} \\ \qquad \mathsf{else}\ exp' \Leftarrow exp : con \rightsquigarrow \\ \quad sbnds_1, \ldots, sbnds_n : \\ \quad sdecs_1, \ldots, sdecs_n\end{array}} \quad (120)$$

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.