A Type-Theoretic Account of Standard ML 1996
(Version 2)

Robert Harper Chris Stone

September 10, 1996

CMU-CS-96-136R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report also appears as Fox Memorandum CMU-CS-FOX-96-02R

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, in part by the National Science Foundation under Grant No. CCR-9502674, and in part 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.

Abstract

A type-theoretic definition of a variant of the Standard ML (Revised 1996) programming language is given. The definition consists of a syntax-directed elaboration of SML '96 programs into a typed intermediate language. The intermediate language is an explicitly-typed λ-calculus with product, sum, recursive, and module types. The translation performs type reconstruction, handles identifier scope resolution, enforces static well-formedness conditions, and expands high-level constructs (such as pattern matching, signature matching, and equality polymorphism) into uses of the more rudimentary mechanisms of the intermediate language.

Version 2 of this document supersedes Version 1 (which was distributed as CMU SCS Technical Report CMU-CS-96-136). This version corrects several errors; it also includes a simpler internal language, an improved presentation of the dynamic semantics, an account of equality polymorphism, and an example extension of the language to handle lazy data structures.

Keywords: Standard ML, type theory, semantics, programming language design, compilation, translation, elaboration, static semantics, dynamic semantics

Contents

1 Introduction

1.1 Overview

This document consists of a type-theoretic account of a variant of Standard ML (Revised 1996) [MTHM96, GR]. The approach taken here is to define the static and dynamic semantics by a syntax-directed translation of the SML '96 abstract syntax (the *external language*, or EL) into an explicitly-typed λ-calculus (the *internal language*, or IL). The translation performs type inference, resolves the scopes of identifiers, enforces static well-formedness conditions, and renders the high-level constructs of the EL as uses of the more rudimentary mechanisms of the IL. The dynamic semantics of the EL is defined by composition of this translation with the dynamic semantics of the IL itself. This is in contrast to the approach initially used in *The Definition of Standard ML* [MTH90] and retained in the revised definition [MTHM96], in which a dynamic semantics is given directly to the raw (untyped) EL independently of the static semantics.

The advantages of this approach are several:

1. The internal language may be shared among many different language definitions given in the style presented here. In particular, we envision the possibility of using the IL given here for the definitions of Scheme and Caml Special Light, perhaps with some relatively minor modifications.

2. The translation given here may be viewed as a reference implementation of a front-end for an SML '96 compiler. In particular we envision using this translation as a guide to the extension of the TIL/ML compiler [HM95, Mor95, Tar96, TMC+96] to SML '96. Compilers for other languages defined by interpretation into the IL can share the back end of the TIL/ML compiler; only a front-end need be written for each specific language.

3. By defining the language in terms of a translation into an intermediate language, we replace the *ad hoc* static semantic objects of *The Definition of Standard ML* by terms of a well-defined typed λ-calculus. This clarifies a number of issues in the design of SML '96. For example, the notion of "type generativity" is replaced by the abstraction mechanisms associated with the formalism of existential types.

There are some disadvantages as well:

1. The translation of datatype declarations is extremely complex. In our view this is a direct reflection of the intrinsic complexity of a mechanism that introduces several mutually recursive abstract types, each a multinary sum of multinary product types.

2. A rigorous definition of the IL must be given in addition to the translation rules from the EL to the IL. This is mitigated somewhat by the possibility of sharing the IL among several language definitions. The burden of defining the IL is shared to some extent by *The Definition*, in the guise of the set of "static semantic objects" and their associated operations.

The type-theoretic definition of SML '96 is divided into three main parts.

Type Structure of the Internal Language. The internal language is an explicitly-typed λ-calculus, with a second-class modules system.

The core of the internal language is based on the XML and λ^{ML} calculi [HM93, HMM90]. The constructors of kind Ω (where Ω is the kind of types) include partial and total function types, record types, sum types, reference types, recursive types, and a single extensible sum. Additionally, the constructors are extended with (restricted) tuples of constructors and functions at the constructor level. Note that there are no polymorphic types (polytypes) in our system.

The modules system is based on the translucent sum or manifest type modules calculi [HL94, Ler94]. In addition to translucent signatures, we have total and partial functor signatures. Our subtyping relation on signatures involves only forgetting of type definitions and totality, and not dropping or reordering components. This means that subtyping has no run-time effect.

The two levels are connected by the ability to define modules local to a core expression.

Dynamic Semantics of the Internal Language. The dynamic semantics is based on a continuation represented as a "stack" of frames, which in total correspond to an evaluation context in a contextual semantics [WF91]. The handling of references and exceptions is similar to Harper's account of polymorphic references [Har93]. We expect this to be much more amenable to direct mechanization than the style of the *Definition*, since there are no implicit rules governing the treatment of exceptions. In fact, the semantics comes very close to defining an abstract machine.

Elaboration. The translation is defined by a series of translation judgments of the general form

$$\Gamma \vdash \textit{EL-phrase} \leadsto \textit{IL-phrase} : \textit{IL-classifier}$$

where Γ is an internal-language context extended with information mapping EL identifiers to IL variables. If Γ^- is the IL context underlying Γ and the above translation judgment holds, we expect the IL typing judgment

$$\Gamma^- \vdash \textit{IL-phrase} : \textit{IL-classifier}$$

also to hold.

1.2 Notes on Implementation

It is important to note that the aim of this document is to define a *language*. We are not advocating that a compiler should generate exactly the same IL code as our elaboration, nor even that the internal language of the compiler should be exactly the IL we present. However, any implementation of an elaborator should be essentially *equivalent* to what we present, and we suggest the compiler's internal language should be *definable* in terms of the IL we present here.

Differences we would expect in an actual implementation include (but are not limited to):

- The IL is likely be extended to make our derived forms into primitives, possibly along with other definable forms (such as a primitive `case` statement for sum types).
- A compiler is likely to use a much more complicated pattern compiler, generating much more efficient pattern-matching code (making use of the primitive `case` statement mentioned above).
- A compiler is likely to use a much more efficient implementation of elaboration contexts, avoiding (but equivalent to!) the the sequential search which we have formalized.
- A compiler is unlikely to use our translation of sequential declarations, which is elegant but which leads to deeply nested structures.

1.3 Major Differences from SML '96

1.3.1 Value Restriction

The value restriction used in this translation is different than that specified in SML '96.

- We treat

  $$\texttt{val}\ \langle\texttt{rec}\rangle\ pat_1\ \texttt{=}\ expr_1\ \texttt{and}\ \ldots\ \texttt{and}\ pat_n\ \texttt{=}\ expr_n$$

  as a derived form for

  $$\texttt{val}\ \langle\texttt{rec}\rangle\ \texttt{(}pat_1\texttt{,}\ldots\texttt{,}pat_n\texttt{)}\ \texttt{=}\ \texttt{(}expr_1\texttt{,}\ldots\texttt{,}expr_n\texttt{)}.$$

  Hence, if any of $expr_1, \ldots, expr_n$ is not a syntactic value, no variables in the entire binding will be generalized. This differs from SSML in which the decision whether to generalize is made on a pattern-by-pattern basis.

- If v is a value, then `val h::t = v` is equivalent to `val h = hd v` and `val t = tl v`. Since in the latter cases `h` and `t` will not be polymorphic (`hd` and `tl` may raise exceptions!), `h` and `t` should not be polymorphic in the former case.

  The translation currently allows variables in a `val` binding to be generalized if the translation of the binding results in a "valuable" structure. At the external language level, this is equivalent to the right-hand side being a value, and no constants or constructors appearing in the pattern, except for the constructor in a single-constructor datatype. (That is, the pattern must be irrefutable.) Note that tuple-patterns pose no problem, since projection is a total function.

  This corresponds to the notion of "polymorphism as substitution," in which

  $$\texttt{let val}\ id\texttt{=}expr\ \texttt{in}\ expr'\ \texttt{end}$$

  is considered to be equivalent to $\{expr/id\}expr'$ when id is polymorphic.

1.3.2 Abstype

We do not consider `abstype` declarations in this document because abstract data types can be easily created using the SML '96 module system instead. There would be no technical difficulties in including `abstype`, if desired.

1.3.3 Laziness

As an example, we show how the translation can be extended (without changing the underlying IL) to incorporate direct support for lazy data structures. We follow the approach suggested by Chris Okasaki [Oka96].

1.3.4 Local and Higher-Order Functors

Our external language permits structure and functor declarations within a `let` or `local` declarations, and also within `structure` and `functor` declarations; this is a reflection of the fact that the internal language has local and higher-order modules.

These higher-order modules are based on the translucent sum formalism [HL94]. As this is compatible with first-class modules (which are not included in this definition), our higher-order functors propagate less type-sharing information than those in the system of MacQueen and Tofte [MT94]. Finding a clean type-theoretic description of this latter system is the subject of current research.

1.3.5 Top-level

Since we allow locally-defined modules in our external language, it suffices to consider programs to be closed expressions of type *ans*, where *ans* is a fixed base type of answers. We do not explicitly consider the translation of SML '96 top-level declarations.

Note that this does not require an implementation to only accept complete programs of some fixed type. For example, a batch compiler may expect a sequence of structure, functor, and signature bindings (SML '96 top-level declarations, possibly separately compiled), which are then "logically" treated as a sequence of declarations in a wrapper generated by the compiler. For a UNIX system, for example, the wrapper might be along the lines of:

```
let ··· in 0 end handle _ => ~1
```

where *ans* is chosen to be the type `int` and a program returns either `0` for normal termination or `~1` for an uncaught exception.

2 Internal Language Abstract Syntax

2.1 Constructors and Kinds

con	$::=$	var	type variables
	$\mid$	$\mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Char} \mid \cdots$	base types
	$\mid$	$\{\mathit{rdecs}\}$	record type
	$\mid$	$\mathit{con}\ \mathsf{Ref}$	reference type
	$\mid$	$\mathit{con} \rightharpoonup \mathit{con}'$	partial function type
	$\mid$	$\mathit{con} \rightarrow \mathit{con}'$	total function type
	$\mid$	Any	extensible sum type
	$\mid$	$\mathit{con}\ \mathsf{Tag}$	exception-tag type
	$\mid$	$\Sigma\,(\mathit{con}_1, \cdots, \mathit{con}_n)$	sum type
	$\mid$	$\mu_i\ \mathit{con}$	recursive type
	$\mid$	$\lambda(\mathit{var}_1, \cdots, \mathit{var}_n).\mathit{con}$	constructor-level function
	$\mid$	$\mathit{con}\ \mathit{con}'$	constructor application
	$\mid$	$(\mathit{con}_1, \cdots, \mathit{con}_n)$	constructor tuple
	$\mid$	$\pi_i\ \mathit{con}$	constructor projection
	$\mid$	$\mathit{mod}_{\mathsf{v}}.\mathit{lbl}$	module projection
rdecs	$::=$	$\cdot$	empty
	$\mid$	$\mathit{rdecs}, \mathit{rdec}$	sequence
rdec	$::=$	$\mathit{lbl} \triangleright \mathit{con}$	record field type
knd	$::=$	Ω^n	kind of n-tuples of types
	$\mid$	$\Omega^m \Rightarrow \Omega^n$	kind of constructor fns.

Figure 1: Constructors and Kinds

Syntactic Restriction: A constructor (con) may only contain at most single total arrow ($\rightarrow$), and if present it must be at the outermost level.

$$
\begin{array}{ll}
\Omega & \mapsto \Omega^1 \\
\mathit{con}_1 \times \cdots \times \mathit{con}_n & \mapsto \{\overline{1} \triangleright \mathit{con}_1, \cdots, \overline{\mathrm{n}} \triangleright \mathit{con}_n\} \\
\mathsf{Unit} & \mapsto \{\} \\
\mathsf{Bool} & \mapsto \Sigma\,(\mathsf{Unit}, \mathsf{Unit})
\end{array}
$$

Figure 2: Derived Forms for Constructors

2.2 Expressions

exp	$::=$	$scon$	constants
	$\mid$	var	variables
	$\mid$	loc	memory locations
	$\mid$	$exp\ exp'$	application
	$\mid$	$\mathsf{fix}\, fbnds\ \mathsf{in}\ var\ \mathsf{end}$	fixed point projection
	$\mid$	$\{rbnds\}$	record expression
	$\mid$	$\pi_{lbl}\ exp$	record projection
	$\mid$	$\mathsf{handle}\ exp\ \mathsf{with}\ exp'$	handle exception
	$\mid$	$\mathsf{raise}^{con}\ exp$	raise exception
	$\mid$	$\mathsf{ref}^{con}\ exp$	allocate new ref cell
	$\mid$	$\mathsf{get}\ exp$	dereference
	$\mid$	$\mathsf{set}\,(exp, exp')$	assignment
	$\mid$	$\mathsf{roll}^{con}\ exp$	coerce into μ type
	$\mid$	$\mathsf{unroll}^{con}\ exp$	coerce from μ type
	$\mid$	$\mathsf{inj}_i^{con}\ exp$	injection into sum
	$\mid$	$\mathsf{proj}_i^{con}\ exp$	projection from sum
	$\mid$	$\mathsf{new\_tag}[con]$	extend type Any
	$\mid$	$\mathsf{tag}_{tag}\ exp$	injection into type Any
	$\mid$	$\mathsf{untag}_{tag}^{con}\ exp$	projection from type Any
	$\mid$	$mod.lbl$	module projection
$rbnds$	$::=$	$\cdot$	empty
	$\mid$	$rbnds, rbnd$	sequence
$rbnd$	$::=$	$lbl \triangleright exp$	record field binding
$fbnds$	$::=$	$\cdot$	empty
	$\mid$	$fbnds, fbnd$	sequence
$fbnd$	$::=$	$var' {=} (var{:}con) {\mapsto} exp$	function binding
$lbls$	$::=$	$lbl \mid lbls.lbl$	sequence of labels
$path$	$::=$	$var \mid var.lbls$	path (qualified variable)

Figure 3: Expressions

$(exp_1, \cdots, exp_n)$	$\mapsto \{\overline{1} \triangleright exp_1, \cdots, \overline{n} \triangleright exp_n\}$
$\lambda var{:}con.exp$	$\mapsto \mathsf{fix}\ var'{=}(var{:}con){\mapsto}exp\ \mathsf{in}\ var'\ \mathsf{end}$
	var' not free in exp
$exp{:}con$	$\mapsto (\lambda var{:}con.var)\ exp$
$exp_1;^{con}\ exp_2$	$\mapsto (\lambda var{:}con.exp_2)\ exp_1$
$exp_1;\ exp_2$	$\mapsto exp_1;^{\mathsf{Unit}}\ exp_2$
$\mathsf{let}\ bnd\ \mathsf{in}\ phrase\ \mathsf{end}$	$\mapsto [1 \triangleright bnd, 2 \triangleright phrase]\ .2$
$\mathsf{failure}$	$\mapsto \mathsf{tag}_{fail}\ \{\}$
fail^{con}	$\mapsto \mathsf{raise}^{con}\ \mathsf{failure}$
$\mathsf{catch}\ exp\ \mathsf{with}\ exp'$	$\mapsto \mathsf{handle}\ exp\ \mathsf{with}\ (\lambda var{:}\mathsf{Any}.$
	$((\mathsf{handle\ untag}^{\mathsf{Unit}}_{fail}\ var\ \mathsf{with}$
	$\lambda var'{:}\mathsf{Any}.\mathsf{raise}^{\mathsf{Unit}}\ var)\ ;$
	$exp'))$
true	$\mapsto \mathsf{inj}^{\mathsf{Bool}}_1\ \{\}$
false	$\mapsto \mathsf{inj}^{\mathsf{Bool}}_2\ \{\}$
$\mathsf{if}^{con}\ exp_1\ \mathsf{then}\ exp_2$	$\mapsto \mathsf{catch}\ (\mathsf{proj}^{\mathsf{Unit}}_1\ exp_1;\ exp_2)\ \mathsf{with}\ exp_3$
$\mathsf{else}\ exp_3$	
$exp_1\ \mathsf{and}\ exp_2$	$\mapsto \mathsf{if}^{\mathsf{Bool}}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else\ false}$
$\mathsf{case}^{\Sigma(con_1, \cdots, con_n)}\ exp\ \mathsf{of}\ exp_1, \cdots, exp_n\ \mathsf{end}$	$\mapsto \mathsf{let}\ var{=}exp\ \mathsf{in}$
	$\mathsf{catch}\ (\mathsf{proj}^{con_1}_1\ var;^{con_1}\ exp_1\ var)\ \mathsf{with} \cdots$
	$\mathsf{catch}\ (\mathsf{proj}^{con_{n-1}}_{n-1}\ var;^{con_{n-1}}\ exp_1\ var)\ \mathsf{with}$
	$exp_n\ var$

Figure 4: Derived Forms for Expressions

We assume there is a distinguished exception tag *fail* of type Unit Tag in the definition of failure. The purpose of the catch form is to serve as a special handler that for *fail* exceptions and to propagate all other exceptions.

2.3 Modules and Signatures

$$
\begin{array}{rcll}
mod & ::= & var & \text{module variable} \\
 & | & [sbnds] & \text{structure} \\
 & | & \lambda var{:}sig.mod & \text{functor} \\
 & | & mod\ mod' & \text{functor application} \\
 & | & mod.lbl & \text{projection from structure} \\
 & | & mod{:}sig & \text{signature ascription} \\
sbnds & ::= & \cdot & \text{structure component bindings} \\
 & | & sbnds, sbnd & \\
sbnd & ::= & lbl \triangleright bnd & \\
bnd & ::= & var{=}con & \text{constructor binding} \\
 & | & var{=}exp & \text{expression binding} \\
 & | & var{=}mod & \text{module binding} \\
sig & ::= & [sdecs] & \text{structure signature} \\
 & | & (var{:}sig){\rightharpoonup}sig' & \text{partial functor signature} \\
 & | & (var{:}sig){\rightarrow}sig' & \text{total functor signature} \\
sdecs & ::= & \cdot & \text{structure declarations} \\
 & | & sdecs, sdec & \\
sdec & ::= & lbl \triangleright dec & \\
decs & ::= & \cdot & \text{declaration lists} \\
 & | & decs, dec & \\
dec & ::= & var{:}con & \text{expression variable declaration} \\
 & | & var{:}sig & \text{module variable declaration} \\
 & | & var{:}knd & \text{opaque type declaration} \\
 & | & var{:}knd{=}con & \text{transparent type declaration} \\
 & | & loc{:}con & \text{typed locations} \\
 & | & tag{:}con & \text{typed exception tag}
\end{array}
$$

Figure 5: Modules and Signatures

- For readability, we often elide the internal names (*var*'s) when writing out *sbnds* (and *sdecs*). In all cases it should be immediately obvious how to consistently restore these with fresh variables.
- We use $sig{\rightharpoonup}sig'$ and $sig{\rightarrow}sig'$ to abbreviate $(var{:}sig){\rightharpoonup}sig'$ and $(var{:}sig){\rightarrow}sig'$ respectively, where var is not free in sig'.

2.4 Useful Abbreviations and Notation

- Many grammatical classes (*lbls*, *decs*, *rdecs*, *rbnds*, *sdecs*, *sbnds*) specify lists of elements. For each of these classes we define a binary append operation, written with a

$$
\begin{array}{rcll}
\mathit{phrase} & ::= & \mathit{exp} & \text{program phrases} \\
& | & \mathit{mod} & \\
& | & \mathit{con} & \\
\\
\mathit{class} & ::= & \mathit{con} & \text{program phrase classifiers} \\
& | & \mathit{sig} & \\
& | & \mathit{knd} &
\end{array}
$$

comma. For example, we define

$$
\begin{array}{rl}
\mathit{decs}, \cdot & := \mathit{decs} \\
\mathit{decs}, (\mathit{decs}', \mathit{dec}') & := (\mathit{decs}, \mathit{decs}'), \mathit{dec}'
\end{array}
$$

with analogous definitions for all the other classes listed above.

- We occasionally use the abbreviation

$$(\mathit{phrase}_i)_{i=1}^{n}$$

as shorthand for

$$\mathit{phrase}_1, \cdots, \mathit{phrase}_n$$

(where the phrases are comma-separated).

2.5 Syntactic Values

$$
\begin{array}{rcl}
exp_{\mathrm{v}} & ::= & scon \\
& \mid & loc \\
& \mid & memo \\
& \mid & path \\
& \mid & \{rbnds_{\mathrm{v}}\} \\
& \mid & \mathsf{fix}\, fbnds\ \mathsf{in}\ var\ \mathsf{end} \\
& \mid & \mathsf{inj}_i^{con}\ exp_{\mathrm{v}} \\
& \mid & \mathsf{tag}_{tag}\ exp_{\mathrm{v}} \\
& \mid & \mathsf{roll}^{con}\ exp_{\mathrm{v}} \\
\\
rbnds_{\mathrm{v}} & ::= & \cdot \\
& \mid & rbnds_{\mathrm{v}}, rbnd_{\mathrm{v}} \\
rbnd_{\mathrm{v}} & ::= & lbl \triangleright exp_{\mathrm{v}} \\
\\
mod_{\mathrm{v}} & ::= & [sbnds_{\mathrm{v}}] \\
& \mid & \lambda var{:}sig.mod \\
bnd_{\mathrm{v}} & ::= & var{=}exp_{\mathrm{v}} \\
& \mid & var{=}mod_{\mathrm{v}} \\
& \mid & var{=}con \\
\\
sbnds_{\mathrm{v}} & ::= & \cdot \\
& \mid & sbnds_{\mathrm{v}}, sbnd_{\mathrm{v}} \\
sbnd_{\mathrm{v}} & ::= & lbl \triangleright bnd_{\mathrm{v}} \\
& & \quad \text{We additionally require these} \\
& & \quad \text{bindings to be } \textit{independent.} \\
\\
val & ::= & exp_{\mathrm{v}} \\
& \mid & mod_{\mathrm{v}} \\
& \mid & con
\end{array}
$$

Figure 6: IL values

We view each class of syntactic values ($class_v$) as subsets of the corresponding class in the abstract syntax ($class$).

2.6 Bindings and Scope

We define the functions BV$(\cdot)$ and dom$(\cdot)$ for various bindings, declarations, and lists thereof:

Function	*Definition*	
$\mathrm{BV}(dec)$	$\mathrm{BV}(var{:}con)$ $\mathrm{BV}(var{:}knd)$ $\mathrm{BV}(var{:}knd{=}con)$ $\mathrm{BV}(var{:}sig)$ $\mathrm{BV}(loc{:}con)$ $\mathrm{BV}(tag{:}con)$	$= var$ $= var$ $= var$ $= var$ $= loc$ $= tag$
$\mathrm{BV}(decs)$	$\mathrm{BV}(dec_1, \cdots, dec_n)$	$= \{\mathrm{BV}(dec_1), \cdots, \mathrm{BV}(dec_n)\}$
$\mathrm{BV}(sdecs)$	$\mathrm{dom}(lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n)$	$= \{\mathrm{BV}(dec_1), \cdots, \mathrm{BV}(dec_n)\}$
$\mathrm{BV}(bnd)$	$\mathrm{BV}(var{=}exp)$ $\mathrm{BV}(var{=}con)$ $\mathrm{BV}(var{=}mod)$	$= var$ $= var$ $= var$
$\mathrm{BV}(sbnds)$	$\mathrm{dom}(lbl_1 \triangleright bnd_1, \cdots, lbl_n \triangleright bnd_n)$	$= \{\mathrm{BV}(bnd_1), \cdots, \mathrm{BV}(bnd_n)\}$
$\mathrm{BV}(\Gamma)$	$\mathrm{BV}(sdec, \Gamma)$ $\mathrm{BV}(lbl \triangleright var{:}\mathsf{Sig}{=}sig, \Gamma)$	$= \{\mathrm{BV}(sdec)\} \cup \mathrm{dom}(\Gamma)$ $= \{var\} \cup \mathrm{dom}(\Gamma)$
$\mathrm{dom}(sdecs)$	$\mathrm{dom}(lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n)$	$= \{lbl_1, \cdots, lbl_n\}$
$\mathrm{dom}(rdecs)$	$\mathrm{dom}(lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n)$	$= \{lbl_1, \cdots, lbl_n\}$
$\mathrm{dom}(\Gamma)$	$\mathrm{dom}(sdec, \Gamma)$ $\mathrm{dom}(lbl \triangleright var{:}\mathsf{Sig}{=}sig, \Gamma)$	$= \{\mathrm{dom}(sdec)\} \cup \mathrm{dom}(\Gamma)$ $= \{lbl\} \cup \mathrm{dom}(\Gamma)$

The scopes of bound variables are given by the following table:

Binding Phrase	*Bound*	Scope
$\mathsf{fix}\, \mathit{fbnds}, \mathit{var}'{=}(\mathit{var}{:}\mathit{con}){\mapsto}\mathit{exp}, \mathit{fbnds}'\ \mathsf{in}\ \mathit{var}''\ \mathsf{end}$ "	var' var	entire phrase exp
$\lambda(\mathit{var}_1, \cdots, \mathit{var}_n).\mathit{con}$	$\mathit{var}_1, \cdots, \mathit{var}_n$	con
$\mathit{sbnd}, \mathit{sbnds}$ $\lambda \mathit{var}{:}\mathit{sig}.\mathit{mod}$	$\mathrm{BV}(\mathit{sbnd})$ var	sbnds mod
$\mathit{sdec}, \mathit{sdecs}$ $\mathit{var}{:}\mathit{sig}{\rightharpoonup}\mathit{sig}'$ $\mathit{var}{:}\mathit{sig}{\rightarrow}\mathit{sig}'$	$\mathrm{BV}(\mathit{sdec})$ var var	sdecs sig' sig'
sdec, Γ $\mathit{lbl}{\triangleright}\mathit{var}{:}\mathsf{Sig}{=}\mathit{sig}, \Gamma$	$\mathrm{BV}(\mathit{sdec})$ var	Γ Γ

We follow standard practice and identify all phrases which differ only with respect to bound variables, locations, and exception names. We use the notation $\mathrm{FV}(\mathit{phrase})$ to denote the set of free expression variables in *phrase*, and $\mathrm{FTV}(\mathit{phrase})$ to denote the set of free type variables in the phrase.

3 Static Semantics

3.1 Introduction

In this section we define the well-formedness and typing judgments for the internal language. Points of interest include:

- There are no "semantic objects" in the sense of the *Definition*.
- The rules are explicitly formulated so that a judgment holds only if its constituents (declaration lists, etc.) are well-formed.
- The set of *valuable* expressions is a class of expression which evaluate to a value without side-effects, referencing the store, or raising exceptions. Similarly, valuable modules are modules which evaluate without side-effects, referencing the store, or raising exceptions. The purpose of distinguishing total and partial functions, as well as total and partial functors, is to specify which function/functor applications are valuable.

3.2 Notation

- Optional elements are enclosed by single brackets $\langle\cdots\rangle$.

3.3 Judgment Forms

Section	*Judgment...*	*Meaning...*
3.4.1	$\vdash \mathit{decs}$ ok	decs is well-formed
	$\mathit{decs} \vdash \mathit{dec}$ ok	dec is well-formed
3.4.2	$\mathit{decs} \vdash \mathit{bnd} : \mathit{dec}$	bnd has declaration dec
3.4.3	$\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind}$	knd is well-formed
3.4.4	$\mathit{decs} \vdash \mathit{con} : \mathit{knd}$	con has kind knd
3.4.5	$\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}$	constructor equivalence at kind knd
3.4.6	$\mathit{decs} \vdash \mathit{exp} : \mathit{con}$	exp has type con
3.4.7	$\mathit{decs} \vdash \mathit{sdecs}$ ok	sdecs is well-formed
	$\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$	sig is well-formed
3.4.8	$\mathit{decs} \vdash \mathit{sdecs} \leq \mathit{sdecs}'$	component-wise subtyping
	$\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}$	signature subtyping
3.4.9	$\mathit{decs} \vdash \mathit{sdecs} \equiv \mathit{sdecs}'$	component-wise equivalence
	$\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig}$	signature equivalence
3.4.10	$\mathit{decs} \vdash \mathit{sbnds} : \mathit{sdecs}$	sbnds has declaration list sdecs
	$\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$	mod has signature sig
3.4.11	$\mathit{decs} \vdash \mathit{exp} \downarrow$	exp is valuable
	$\mathit{decs} \vdash \mathit{mod} \downarrow$	mod is valuable
	$\mathit{decs} \vdash \mathit{exp} \downarrow \mathit{con}$	exp is valuable with type con
	$\mathit{decs} \vdash \mathit{mod} \downarrow \mathit{sig}$	mod is valuable with signature sig

3.4 Inference Rules

3.4.1 Well-formed Declarations

$$\boxed{\vdash \mathit{decs}\ \mathbf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathrm{ok}} \tag{1}$$

$$\frac{\mathit{decs} \vdash \mathit{dec}\ \mathrm{ok}}{\vdash \mathit{decs}, \mathit{dec}\ \mathrm{ok}} \tag{2}$$

Rules 1 and 2: Via the coercion of translation contexts into declaration lists, this judgment induces the judgment $\vdash \Gamma$ ok.

$$\boxed{\mathit{decs} \vdash \mathit{dec}\ \mathbf{ok}}$$

$$\frac{\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{knd}\ \mathrm{ok}} \tag{3}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{knd}{=}\mathit{con}\ \mathrm{ok}} \tag{4}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \Omega \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{con}\ \mathrm{ok}} \tag{5}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \qquad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}\ \mathrm{ok}} \tag{6}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}'\ \mathsf{Ref} : \Omega \qquad \mathit{loc} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{loc}{:}\mathit{con}\ \mathrm{ok}} \tag{7}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}'\ \mathsf{Tag} : \Omega \qquad \mathit{tag} \notin \mathrm{BV}(\mathit{decs})}{\mathit{decs} \vdash \mathit{tag}{:}\mathit{con}\ \mathrm{ok}} \tag{8}$$

3.4.2 Well-formed Bindings

$$\boxed{\mathit{decs} \vdash \mathit{bnd}\ :\ \mathit{dec}}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathit{knd}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{con}\ :\ \mathit{var}{:}\mathit{knd}} \tag{9}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{con}\ :\ \mathit{var}{:}\mathit{knd}{=}\mathit{con}'} \tag{10}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{exp}\ :\ \mathit{var}{:}\mathit{con}} \tag{11}$$

$$\frac{\mathit{decs} \vdash \mathit{mod} : \mathit{sig}' \qquad \mathit{decs} \vdash \mathit{sig}' \leq \mathit{sig} : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{=}\mathit{mod}\ :\ \mathit{var}{:}\mathit{sig}} \tag{12}$$

3.4.3 Well-formed Kinds

$$\boxed{decs \vdash knd : \mathsf{Kind}}$$

$$\frac{n \geq 0}{decs \vdash \Omega^n : \mathsf{Kind}} \tag{13}$$

$$\frac{n, m \geq 0}{decs \vdash \Omega^m \Rightarrow \Omega^n : \mathsf{Kind}} \tag{14}$$

3.4.4 Well-formed Constructors

$$\boxed{decs \vdash con : knd}$$

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

$$\frac{}{decs \vdash \mathsf{Any} : \Omega} \tag{16}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con\, \mathsf{Ref} : \Omega} \tag{17}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con\, \mathsf{Tag} : \Omega} \tag{18}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \rightharpoonup con' : \Omega} \tag{19}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \to con' : \Omega} \tag{20}$$

$$\frac{lbl_1, \cdots, lbl_n \text{ distinct} \qquad decs \vdash con_1 : \Omega \quad \cdots \quad decs \vdash con_n : \Omega}{decs \vdash \{lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n\} : \Omega} \tag{21}$$

$$\frac{decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n}{decs \vdash \mu_i\, con : \Omega} \tag{22}$$

$$\frac{decs, var_1{:}\Omega, \cdots, var_n{:}\Omega \vdash con : \Omega^m}{decs \vdash \lambda(var_1, \cdots, var_n).con : \Omega^n \Rightarrow \Omega^m} \tag{23}$$

$$\frac{decs \vdash con_1 : \Omega \qquad \cdots \qquad decs \vdash con_n : \Omega}{decs \vdash \Sigma\,(con_1, \cdots, con_n) : \Omega} \tag{24}$$

$$\frac{decs \vdash con_1 : \Omega \qquad \cdots \qquad decs \vdash con_n : \Omega}{decs \vdash (con_1, \cdots, con_n) : \Omega^n} \tag{25}$$

$$\frac{decs \vdash con : knd' \Rightarrow knd \qquad decs \vdash con : knd'}{decs \vdash con\ con' : knd} \tag{26}$$

$$\frac{decs \vdash mod_{\mathrm{v}} : [sdecs, lbl \triangleright var{:}knd\langle = con' \rangle, sdecs']}{decs \vdash mod_{\mathrm{v}}.lbl : knd} \tag{27}$$

3.4.5 Constructor Equivalence

$$\boxed{decs \vdash con \equiv con' : knd}$$

$$\frac{\begin{array}{c}\vdash decs \text{ ok} \\ decs = decs', var{:}knd{=}con, decs''\end{array}}{decs \vdash var \equiv con : knd} \tag{28}$$

Rule 28: The well-formedness judgment $\vdash decs$ ok requires that all the variables bound in the list must be distinct; there can be no variable capture when bringing *con* into the scope of the variables in $decs''$.

$$\frac{\begin{array}{c}decs \vdash mod_{\mathrm{v}} : [lbl_1 \triangleright dec_1, \cdots, lbl_m \triangleright dec_m, \\ \qquad lbl \triangleright var{:}knd{=}con', sdecs] \\ decs, dec_1, \cdots, dec_m \vdash con \equiv con' : knd \\ decs \vdash con : knd\end{array}}{decs \vdash mod_{\mathrm{v}}.lbl \equiv con : knd} \tag{29}$$

Rule 29: The projection must yield a valid constructor with respect to the ambient context.

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \rightharpoonup con_1' \equiv con_2 \rightharpoonup con_2' : \Omega} \tag{30}$$

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \rightarrow con_1' \equiv con_2 \rightarrow con_2' : \Omega} \tag{31}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con\ \mathsf{Ref} \equiv con'\ \mathsf{Ref} : \Omega} \tag{32}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con\ \mathsf{Tag} \equiv con'\ \mathsf{Tag} : \Omega} \tag{33}$$

$$\frac{\begin{array}{c}lbl_1, \cdots, lbl_n \text{ distinct} \\ decs \vdash con_1 \equiv con_1' : \Omega \quad \cdots \quad decs \vdash con_n \equiv con_n' : \Omega\end{array}}{\begin{array}{c}decs \vdash \{lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n\} \equiv \\ \qquad \{lbl_1 \triangleright con_1', \cdots, lbl_n \triangleright con_n'\} : \Omega\end{array}} \tag{34}$$

Rule 34: To be equivalent, two IL record types must have equivalent components with the same labels in the same order.

$$\frac{\begin{array}{c} i \in 1..n \\ decs \vdash con \equiv con' : \Omega^n \Rightarrow \Omega^n \end{array}}{decs \vdash \mu_i\ con \equiv \mu_i\ con' : \Omega} \tag{35}$$

$$\frac{decs, var_1{:}\Omega, \cdots, var_n{:}\Omega \vdash con \equiv con' : \Omega^m}{\begin{array}{l} decs \vdash \lambda(var_1, \cdots, var_n).con \\ \qquad \equiv \lambda(var_1, \cdots, var_n).con' : \Omega^n \Rightarrow \Omega^m \end{array}} \tag{36}$$

$$\frac{decs \vdash con_1 \equiv con_1' : \Omega \quad \cdots \quad decs \vdash con_n \equiv con_n' : \Omega}{decs \vdash \Sigma\,(con_1, \cdots, con_n) \equiv \Sigma\,(con_1', \cdots, con_n') : \Omega} \tag{37}$$

$$\frac{decs \vdash con_1 \equiv con_1' : \Omega \quad \cdots \quad decs \vdash con_n \equiv con_n' : \Omega}{decs \vdash (con_1, \cdots, con_n) \equiv (con_1', \cdots, con_n') : \Omega^n} \tag{38}$$

$$\frac{\begin{array}{c} decs \vdash con_1 \equiv con_1' : knd' \Rightarrow knd \\ decs \vdash con_2 \equiv con_2' : knd' \end{array}}{decs \vdash con_1\ con_2 \equiv con_1'\ con_2' : knd} \tag{39}$$

$$\frac{\begin{array}{c} decs, var_1{:}\Omega, \cdots, var_n{:}\Omega \vdash con : \Omega^m \\ decs \vdash con' \equiv (con_1', \cdots, con_n') : \Omega^n \end{array}}{\begin{array}{l} decs \vdash (\lambda(var_1, \cdots, var_n).con)\ con' \\ \qquad \equiv \{con_1'/var_1\} \cdots \{con_n'/var_n\} con : \Omega^m \end{array}} \tag{40}$$

$$\frac{\begin{array}{c} var_1, \cdots, var_n \notin \mathrm{BV}(decs) \\ decs \vdash con : \Omega^n \Rightarrow \Omega^m \end{array}}{\begin{array}{l} decs \vdash \lambda(var_1, \cdots, var_n).(con\,(var_1, \cdots, var_n)) \\ \qquad \equiv con : \Omega^m \end{array}} \tag{41}$$

$$\frac{\begin{array}{c} i \in 1..n \\ decs \vdash con \equiv (con_1, \cdots, con_n) : \Omega^n \end{array}}{decs \vdash \pi_i\ con \equiv con_i : \Omega} \tag{42}$$

$$\frac{decs \vdash con : \Omega^n}{decs \vdash (\pi_1\ con, \cdots, \pi_n\ con) \equiv con : \Omega^n} \tag{43}$$

$$\frac{decs \vdash con : knd}{decs \vdash con \equiv con : knd} \tag{44}$$

$$\frac{decs \vdash con' \equiv con : knd}{decs \vdash con \equiv con' : knd} \tag{45}$$

$$\frac{\begin{array}{c} decs \vdash con \equiv con' : knd \\ decs \vdash con' \equiv con'' : knd \end{array}}{decs \vdash con \equiv con'' : knd} \tag{46}$$

3.4.6 Well-formed Expressions

$$\boxed{decs \vdash exp : con}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', var{:}con, decs''}{decs \vdash var : con} \tag{47}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', loc{:}con, decs''}{decs \vdash loc : con} \tag{48}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', tag{:}con, decs''}{decs \vdash tag : con} \tag{49}$$

$$\frac{decs \vdash exp : con' \rightharpoonup con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \tag{50}$$

$$\frac{\begin{array}{c} k \in 1..n \\ decs, (var'_i{:}con_i \rightharpoonup con'_i)_{i=1}^n, var_1{:}con_1 \vdash exp_1 : con'_1 \\ \vdots \\ decs, (var'_i{:}con_i \rightharpoonup con'_i)_{i=1}^n, var_n{:}con_n \vdash exp_n : con'_n \end{array}}{decs \vdash \mathsf{fix}\,(var'_i{=}(var_i{:}con_i){\mapsto}exp_i)_{i=1}^n \;\mathsf{in}\; var'_k \;\mathsf{end} : con_k \rightharpoonup con'_k} \tag{51}$$

$$\frac{\begin{array}{c} k \in 1..n \\ var'_1, \cdots, var'_n \notin \mathrm{BV}(decs) \\ decs, var_k{:}con_k \vdash exp_k \downarrow con'_k \end{array}}{decs \vdash \mathsf{fix}\,(var'_i{=}(var_i{:}con_i){\mapsto}exp_i)_{i=1}^n \;\mathsf{in}\; var'_k \;\mathsf{end} : con_k \rightarrow con'_k} \tag{52}$$

Rule 52: To guarantee that total functions always terminate, we make the strong restriction that a total function not call any function (even total functions) in the mutually-recursive set.

$$\frac{\begin{array}{c} decs \vdash exp_1 : con_1 \\ \vdots \\ decs \vdash exp_n : con_n \\ lbl_1, \cdots, lbl_n \text{ distinct} \end{array}}{decs \vdash \{lbl_1 \triangleright exp_1, \cdots, lbl_n \triangleright exp_n\} : \{lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n\}} \tag{53}$$

Rule 53: For the record bindings to be well-typed with respect to the declarations, they must have the same labels in the same order (with no duplicate labels) and components must be well-typed elementwise.

$$\frac{decs \vdash exp : \{rdecs, lbl \triangleright con, rdecs'\}}{decs \vdash \pi_{lbl}\ exp : con} \tag{54}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp' : \mathsf{Any} \rightharpoonup con}{decs \vdash \mathsf{handle}\ exp\ \mathsf{with}\ exp' : con} \tag{55}$$

$$\frac{decs \vdash exp : \mathsf{Any}}{decs \vdash \mathsf{raise}^{con}\ exp : con} \tag{56}$$

$$\frac{decs \vdash con : \Omega}{\begin{array}{l} decs \vdash \mathsf{new\_tag}[con] : \\ \qquad \{\mathrm{mk}{:}con \to \mathsf{Any}, \mathrm{km}{:}\mathsf{Any} \rightharpoonup con\} \end{array}} \tag{57}$$

Rule 57: The evaluation of new_tag[] creates a new exception tag, and creates a record containing the corresponding tagging and untagging functions.

$$\frac{decs \vdash exp : con}{decs \vdash \mathsf{ref}^{con}\ exp : con\ \mathsf{Ref}} \tag{58}$$

$$\frac{decs \vdash exp : con\ \mathsf{Ref}}{decs \vdash \mathsf{get}\ exp : con} \tag{59}$$

$$\frac{\begin{array}{c} decs \vdash exp : con\ \mathsf{Ref} \\ decs \vdash exp' : con \end{array}}{decs \vdash \mathsf{set}\,(exp, exp') : \{\}} \tag{60}$$

$$\frac{\begin{array}{c} decs \vdash con' \equiv \mu_i\ con : \Omega \\ decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n \\ decs \vdash exp : \pi_i\,(con\,(\mu_1\ con, \cdots, \mu_n\ con)) \end{array}}{decs \vdash \mathsf{roll}^{con'}\ exp : \mu_i\ con} \tag{61}$$

$$\frac{\begin{array}{c} decs \vdash con' \equiv \mu_i\ con : \Omega \\ decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n \\ decs \vdash exp : \mu_i\ con \end{array}}{decs \vdash \mathsf{unroll}^{con'}\ exp : \pi_i\,(con\,(\mu_1\ con, \cdots, \mu_n\ con))} \tag{62}$$

$$\frac{\begin{array}{c} decs \vdash con \equiv \Sigma\,(con_1, \cdots, con_n) : \Omega \\ 1 \leq i \leq n \qquad decs \vdash exp : con_i \end{array}}{decs \vdash \mathsf{inj}_i^{con}\ exp : con} \tag{63}$$

$$\frac{decs \vdash exp : \Sigma\,(con_1, \cdots, con_n) \qquad 1 \leq i \leq n}{decs \vdash \mathsf{proj}_i^{con_i}\ exp : con} \tag{64}$$

$$\frac{decs \vdash tag : con\ \mathsf{Tag} \qquad decs \vdash exp : con}{decs \vdash \mathsf{tag}_{tag}\ exp : \mathsf{Any}} \tag{65}$$

$$\frac{decs \vdash tag : con\ \mathsf{Tag} \qquad decs \vdash exp : \mathsf{Any}}{decs \vdash \mathsf{untag}^{con}_{tag}\ exp : con} \tag{66}$$

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \cdots, lbl_m \triangleright dec_m, \\ \quad lbl \triangleright var{:}con', sdecs] \\ decs, dec_1, \cdots, dec_m \vdash con \equiv con' : \Omega \\ decs \vdash con : \Omega \end{array}}{decs \vdash mod.lbl : con} \tag{67}$$

Rule 67: A projection is only well-formed if the result is typable. If *mod* is a value, the projection is always well-formed.

$$\frac{decs \vdash exp : con \to con'}{decs \vdash exp : con \rightharpoonup con'} \tag{68}$$

Rule 68: Since we only allow a total arrow at the outermost level of a constructor, this rule suffices to completely describe the "subtyping" of constructors.

$$\frac{decs \vdash exp : con' \qquad decs \vdash con \equiv con' : \Omega}{decs \vdash exp : con} \tag{69}$$

3.4.7 Well-formed Signatures

$$\boxed{decs \vdash sdecs\ \textbf{ok}}$$

$$\frac{\vdash decs\ \mathrm{ok}}{decs \vdash \cdot\ \mathrm{ok}} \tag{70}$$

$$\frac{\begin{array}{c} decs \vdash dec\ \mathrm{ok} \\ decs, dec \vdash sdecs\ \mathrm{ok} \\ lbl \notin \mathrm{dom}(sdecs) \end{array}}{decs \vdash lbl \triangleright dec, sdecs\ \mathrm{ok}} \tag{71}$$

$$\boxed{decs \vdash sig : \mathsf{Sig}}$$

$$\frac{decs \vdash sdecs\ \mathrm{ok}}{decs \vdash [sdecs] : \mathsf{Sig}} \tag{72}$$

$$\frac{decs, var{:}sig \vdash sig' : \mathsf{Sig}}{decs \vdash var{:}sig \rightharpoonup sig' : \mathsf{Sig}} \tag{73}$$

$$\frac{decs, var{:}sig \vdash sig' : \mathsf{Sig}}{decs \vdash var{:}sig \to sig' : \mathsf{Sig}} \tag{74}$$

3.4.8 Signature Subtyping

$$\boxed{\mathit{decs} \vdash \mathit{sdecs} \ \leq \ \mathit{sdecs}'}$$

$$\frac{}{\mathit{decs} \vdash \cdot \ \leq \ \cdot} \tag{75}$$

$$\frac{\mathit{decs}, \mathit{var}{:}\mathit{knd}{=}\mathit{con} \vdash \mathit{sdecs} \ \leq \ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs} \ \leq \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}'} \tag{76}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig} \\ \mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sdecs} \ \leq \ \mathit{sdecs}'\end{array}}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs} \ \leq \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'} \tag{77}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{dec} \ \equiv \ \mathit{lbl}{\triangleright}\mathit{dec}' \\ \mathit{decs}, \mathit{dec} \vdash \mathit{sdecs} \ \leq \ \mathit{sdecs}'\end{array}}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{dec}, \mathit{sdecs} \ \leq \ \mathit{lbl}{\triangleright}\mathit{dec}', \mathit{sdecs}'} \tag{78}$$

$$\boxed{\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}}$$

$$\frac{\mathit{decs} \vdash \mathit{sdecs} \ \leq \ \mathit{sdecs}'}{\mathit{decs} \vdash [\mathit{sdecs}] \leq [\mathit{sdecs}'] : \mathsf{Sig}} \tag{79}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 {\rightharpoonup} \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 {\rightharpoonup} \mathit{sig}_2' : \mathsf{Sig}} \tag{80}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 {\rightarrow} \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 {\rightharpoonup} \mathit{sig}_2' : \mathsf{Sig}} \tag{81}$$

$$\frac{\mathit{decs} \vdash \mathit{sig}_2 \leq \mathit{sig}_1 : \mathsf{Sig} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig}_2 \vdash \mathit{sig}_1' \leq \mathit{sig}_2' : \mathsf{Sig}}{\mathit{decs} \vdash \mathit{var}{:}\mathit{sig}_1 {\rightarrow} \mathit{sig}_1' \leq \mathit{var}{:}\mathit{sig}_2 {\rightarrow} \mathit{sig}_2' : \mathsf{Sig}} \tag{82}$$

3.4.9 Signature Equivalence

$$\boxed{\mathit{decs} \vdash \mathit{sdecs} \ \equiv \ \mathit{sdecs}'}$$

$$\frac{}{\mathit{decs} \vdash \cdot \ \equiv \ \cdot} \tag{83}$$

$$\frac{\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind} \quad \mathit{decs}, \mathit{var}{:}\mathit{knd} \vdash \mathit{sdecs} \ \equiv \ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs} \ \equiv \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}, \mathit{sdecs}'} \tag{84}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \quad \mathit{decs}, \mathit{var}{:}\mathit{knd}{=}\mathit{con} \vdash \mathit{sdecs} \ \equiv \ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}, \mathit{sdecs} \ \equiv \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{knd}{=}\mathit{con}', \mathit{sdecs}'} \tag{85}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \Omega \quad \mathit{decs}, \mathit{var}{:}\mathit{con} \vdash \mathit{sdecs} \ \equiv \ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{con}, \mathit{sdecs} \ \equiv \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{con}', \mathit{sdecs}'} \tag{86}$$

$$\frac{\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathit{knd} \quad \mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sdecs} \ \equiv \ \mathit{sdecs}'}{\mathit{decs} \vdash \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{sig}, \mathit{sdecs} \ \equiv \ \mathit{lbl}{\triangleright}\mathit{var}{:}\mathit{sig}', \mathit{sdecs}'} \tag{87}$$

$$\boxed{decs \vdash sig \equiv sig' : \mathsf{Sig}}$$

$$\frac{decs \vdash sdecs \ \equiv \ sdecs'}{decs \vdash [sdecs] \equiv [sdecs'] : \mathsf{Sig}} \tag{88}$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \mathsf{Sig} \qquad decs, var{:}sig_1 \vdash sig'_1 \equiv sig'_2 : \mathsf{Sig}}{decs \vdash var{:}sig_1 \rightharpoonup sig'_1 \equiv var{:}sig_2 \rightharpoonup sig'_2 : \mathsf{Sig}} \tag{89}$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \mathsf{Sig} \qquad decs, var{:}sig_1 \vdash sig'_1 \equiv sig'_2 : \mathsf{Sig}}{decs \vdash var{:}sig_1 \rightarrow sig'_1 \equiv var{:}sig_2 \rightarrow sig'_2 : \mathsf{Sig}} \tag{90}$$

3.4.10 Well-formed Modules

$$\boxed{decs \vdash sbnds \ : \ sdecs}$$

$$\frac{}{decs \vdash \cdot \ : \ \cdot} \tag{91}$$

$$\frac{decs \vdash bnd \ : \ dec \qquad decs, dec \vdash sbnds \ : \ sdecs}{decs \vdash lbl \triangleright bnd, sbnds \ : \ lbl \triangleright dec, sdecs} \tag{92}$$

$$\boxed{decs \vdash mod : sig}$$

$$\frac{\begin{array}{c}\vdash decs \ \mathrm{ok} \\ decs = decs', var{:}sig, decs''\end{array}}{decs \vdash var : sig} \tag{93}$$

$$\frac{decs \vdash sbnds \ : \ sdecs}{decs \vdash [sbnds] : [sdecs]} \tag{94}$$

$$\frac{decs, var{:}sig \vdash mod : sig'}{decs \vdash \lambda var{:}sig.mod : var{:}sig \rightharpoonup sig'} \tag{95}$$

$$\frac{decs, var{:}sig \vdash mod \downarrow sig'}{decs \vdash \lambda var{:}sig.mod : var{:}sig \rightarrow sig'} \tag{96}$$

$$\frac{decs \vdash mod : sig' \rightharpoonup sig \qquad decs \vdash mod' : sig'}{decs \vdash mod\ mod' : sig} \tag{97}$$

Rule 97: Only functors with non-dependent types may be applied. Dependencies can be eliminated by uses of the subtyping and equivalence rules. If the argument is a value, dependencies can always be eliminated.

$$\frac{\begin{array}{c}decs \vdash mod : [lbl_1 \triangleright dec_1, \cdots, lbl_m \triangleright dec_m, \\ \qquad lbl \triangleright var{:}sig', sdecs] \\ decs, dec_1, \cdots, dec_m \vdash sig' \leq sig : \mathsf{Sig} \\ decs \vdash sig : \mathsf{Sig}\end{array}}{decs \vdash mod.lbl : sig} \tag{98}$$

Rule 98: A projection is only well-formed if the result can be given a signature in the ambient context. If *mod* is a value, the projection is always well-formed.

$$\frac{decs \vdash mod : sig}{decs \vdash mod{:}sig : sig} \tag{99}$$

Rule 99: Ascription of a signature to a module can permanently forget type definitions visible in *mod*.

$$\frac{decs \vdash mod_{\mathrm{v}} : [sdecs, lbl \triangleright var{:}knd, sdecs']}{decs \vdash mod_{\mathrm{v}} : [sdecs, lbl \triangleright var{:}knd{=}mod_{\mathrm{v}}.lbl, sdecs']} \tag{100}$$

Rule 100: The "self" rule. If *mod.lbls* specifies a type and *mod* has a well-defined value then $mod.lbls \equiv mod.lbls$; we add this fact to the signature.

$$\frac{\begin{array}{c} decs \vdash mod_{\mathrm{v}} : [sdecs, lbl \triangleright var{:}sig, sdecs'] \\ decs \vdash mod_{\mathrm{v}}.lbl : sig' \end{array}}{decs \vdash mod_{\mathrm{v}} : [sdecs, lbl \triangleright var{:}sig', sdecs']} \tag{101}$$

Rule 101: Allows the "self" rule to be applied to substructures.

$$\frac{decs \vdash mod : sig' \qquad decs \vdash sig' \leq sig : \mathsf{Sig}}{decs \vdash mod : sig} \tag{102}$$

3.4.11 Valuability

$$\boxed{decs \vdash exp \downarrow con}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \tag{103}$$

$$\boxed{decs \vdash mod \downarrow sig}$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \tag{104}$$

$$\boxed{decs \vdash exp \downarrow}$$

$$\frac{}{decs \vdash exp_{\mathrm{v}} \downarrow} \tag{105}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \tag{106}$$

$$\frac{decs \vdash exp_1 \downarrow con' \to con \qquad decs \vdash exp_2 \downarrow}{decs \vdash exp_1\ exp_2 \downarrow} \tag{107}$$

$$\frac{decs \vdash exp_1 \downarrow \qquad \cdots \qquad decs \vdash exp_n \downarrow}{decs \vdash \{lbl_1 \triangleright exp_1, \cdots, lbl_n \triangleright exp_n\} \downarrow} \tag{108}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \pi_{lbl}\ exp \downarrow} \tag{109}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{roll}^{con}\ exp \downarrow} \tag{110}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{unroll}^{con}\ exp \downarrow} \tag{111}$$

$$\frac{decs \vdash exp \downarrow \Sigma\,(con)}{decs \vdash \mathsf{proj}_1^{con}\ exp \downarrow} \tag{112}$$

Rule 112: Projection from a single-element sum type is total.

$$\boxed{decs \vdash sbnds \downarrow}$$

$$\frac{}{decs \vdash \cdot \downarrow} \tag{113}$$

$$\frac{decs \vdash exp \downarrow con \qquad decs, var{:}con \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var{=}exp, sbnds \downarrow} \tag{114}$$

$$\frac{decs \vdash con : knd \qquad decs, var{:}knd{=}con \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var{=}con, sbnds \downarrow} \tag{115}$$

$$\frac{decs \vdash mod \downarrow sig \qquad decs, var{:}sig \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var{=}mod, sbnds \downarrow} \tag{116}$$

$$\boxed{decs \vdash mod \downarrow}$$

$$\frac{}{decs \vdash mod_{\mathrm{v}} \downarrow} \tag{117}$$

$$\frac{decs \vdash sbnds \downarrow}{decs \vdash [sbnds] \downarrow} \tag{118}$$

$$\frac{decs \vdash mod \downarrow sig' \to sig \qquad decs \vdash mod' \downarrow}{decs \vdash mod\ mod' \downarrow} \tag{119}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \tag{120}$$

4 Dynamic Semantics

4.1 Introduction

The dynamic semantics of the internal language is a call-by-value operational semantics presented as a rewriting system on states of an abstract machine. The presentation is strongly influenced by the work of Plotkin [Plo81] and Wright and Felleisen [WF91], and is a significant departure from the framework employed in *The Definition.* In particular we employ a *small step* semantics in which transitions represent basic evaluation steps of an abstract machine. We rely on substitution, rather than environments; values are particular expressions of the language. Exception propagation is handled by explicitly maintaining the evaluation context, rather than relying on implicit rules for exception propagation. We maintain a store for assignable cells, as in *The Definition*, and a typing context which types locations and dynamically-created tags associated with values of type Any; these are the only extra-linguistic constructs in the formalism. As Wright and Felleisen have demonstrated these could be made part of the language by introducing a `letref` construct (at the module level).

Each state Σ of the abstract machine is a four-tuple of the form

$$(\Delta, \sigma, E, \mathit{phrase}),$$

where

- Δ is a typing context (*decs*) for locations and tags created at run-time. This maintains a record of what exception tags and locations have been allocated, and is also used in our soundness proofs.
- σ is a finite mapping from locations (*loc*'s) typed in Δ to expression values ($\mathit{exp}_{\mathrm{v}}$).
- E is a stack of evaluation frames (see Figure 7). This represents an implementation's control stack, or equivalently, the current continuation. Appending two stacks corresponds to composition of continuations; accordingly, we write $E \circ E'$ for the concatenation of E and E', defined in the obvious way. The meta-variable RS ranges over control stacks that do not contain a frame of the form "$\mathsf{handle}\,[]\,\mathsf{with}\ \mathit{exp}$".
- *phrase* is an expression, module, or constructor.

We let Σ_t range over the set of terminal states, where a state is called *terminal* if it has one of the forms:

$(\Delta, \sigma, [], \mathit{val})$	normal termination
$(\Delta, \sigma, [], \mathsf{raise}^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}})$	uncaught exception

All other states are called *nonterminal.*

The dynamic semantics is a transition relation $\Sigma \hookrightarrow \Sigma'$ between states. As usual, we denote the reflexive, transitive closure of $\hookrightarrow$ by $\hookrightarrow^*$.

$$
\begin{array}{rcl}
F & ::= & [\,]\ exp \\
& | & exp_{\mathrm{v}}\ [\,] \\
& | & \{rbnds_{\mathrm{v}}, lbl \triangleright [\,], rbnds\} \\
& | & \pi_{lbl}\ [\,] \\
& | & \mathsf{handle}\ [\,]\ \mathsf{with}\ exp \\
& | & \mathsf{raise}^{con}\ [\,] \\
& | & \mathsf{ref}^{con}\ [\,] \\
& | & \mathsf{get}\ [\,] \\
& | & \mathsf{set}\ ([\,], exp) \\
& | & \mathsf{set}\ (exp_{\mathrm{v}}, [\,]) \\
& | & \mathsf{roll}^{con}\ [\,] \\
& | & \mathsf{unroll}^{con}\ [\,] \\
& | & \mathsf{inj}_i^{con}\ [\,] \\
& | & \mathsf{proj}_i^{con}\ [\,] \\
& | & \mathsf{tag}_{tag}\ [\,] \\
& | & \mathsf{untag}_{tag}^{con}\ [\,] \\
& | & [sbnds_{\mathrm{v}}, lbl \triangleright var {=} [\,], sbnds] \\
& | & [\,]\ mod \\
& | & mod_{\mathrm{v}}\ [\,] \\
& | & [\,].lbl \\
& | & [\,]{:}sig \\
\\
E & ::= & [\,] \\
& | & E \circ F
\end{array}
$$

Figure 7: Evaluation Frames

4.2 Transition Relation

4.2.1 Search for Redex

$$(\Delta, \sigma, E, exp\ exp') \hookrightarrow (\Delta, \sigma, E \circ ([\,]\ exp'), exp) \tag{121}$$

$$(\Delta, \sigma, E \circ [[\,]\ exp], exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E \circ (exp_{\mathrm{v}}\ [\,]), exp) \tag{122}$$

$$\begin{array}{l}(\Delta, \sigma, E, \{rbnds_{\mathrm{v}}, lbl \triangleright exp, rbnds\}) \hookrightarrow \\ \quad (\Delta, \sigma, E \circ \{rbnds_{\mathrm{v}}, lbl \triangleright [\,], rbnds\}, exp) \\ \qquad\qquad \text{if } exp \text{ is not a value}\end{array} \tag{123}$$

$$\begin{array}{l}(\Delta, \sigma, E \circ \{rbnds_{\mathrm{v}}, lbl \triangleright [\,], rbnds_{\mathrm{v}}'\}, exp_{\mathrm{v}}) \hookrightarrow \\ \quad (\Delta, \sigma, E, \{rbnds_{\mathrm{v}}, lbl \triangleright exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\})\end{array} \tag{124}$$

$$\begin{array}{l}(\Delta, \sigma, E \circ \{rbnds_{\mathrm{v}}, lbl \triangleright [\,], rbnds_{\mathrm{v}}', lbl' \triangleright exp', rbnds\}, exp_{\mathrm{v}}) \hookrightarrow \\ \quad (\Delta, \sigma, E \circ \{rbnds_{\mathrm{v}}, lbl \triangleright exp_{\mathrm{v}}, rbnds_{\mathrm{v}}', lbl' \triangleright [\,], rbnds\}, exp') \\ \qquad\qquad \text{if } exp' \text{ is not a value}\end{array} \tag{125}$$

$$(\Delta, \sigma, E, \pi_{\mathit{lbl}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\pi_{\mathit{lbl}}\ []), \mathit{exp}) \tag{126}$$

$$\begin{array}{l}(\Delta, \sigma, E, \textsf{handle}\ \mathit{exp}\ \textsf{with}\ \mathit{exp}') \hookrightarrow \\ \quad (\Delta, \sigma, E \circ (\textsf{handle}\ []\ \textsf{with}\ \mathit{exp}'), \mathit{exp})\end{array} \tag{127}$$

$$\begin{array}{r}(\Delta, \sigma, E, \textsf{raise}^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{raise}^{\mathit{con}}\ []), \mathit{exp}) \\ \text{if } \mathit{exp} \text{ is not a value}\end{array} \tag{128}$$

$$(\Delta, \sigma, E \circ (\textsf{raise}^{\mathit{con}}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \textsf{raise}^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}}) \tag{129}$$

$$(\Delta, \sigma, E, \textsf{ref}^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{ref}^{\mathit{con}}\ []), \mathit{exp}) \tag{130}$$

$$(\Delta, \sigma, E, \textsf{get}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{get}\ []), \mathit{exp}) \tag{131}$$

$$\begin{array}{l}(\Delta, \sigma, E, \textsf{set}\ (\mathit{exp}, \mathit{exp}')) \hookrightarrow \\ \quad (\Delta, \sigma, E \circ (\textsf{set}\ ([], \mathit{exp}')), \mathit{exp})\end{array} \tag{132}$$

$$\begin{array}{l}(\Delta, \sigma, E \circ (\textsf{set}\ ([], \mathit{exp})), \mathit{exp}_{\mathrm{v}}) \hookrightarrow \\ \qquad\qquad (\Delta, \sigma, (\textsf{set}\ (\mathit{exp}_{\mathrm{v}}, [])), \mathit{exp})\end{array} \tag{133}$$

$$(\Delta, \sigma, E, \textsf{roll}^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{roll}^{\mathit{con}}\ []), \mathit{exp}) \tag{134}$$

$$(\Delta, \sigma, E \circ (\textsf{roll}^{\mathit{con}}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \textsf{roll}^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}}) \tag{135}$$

$$(\Delta, \sigma, E, \textsf{unroll}^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{unroll}^{\mathit{con}}\ []), \mathit{exp}) \tag{136}$$

$$(\Delta, \sigma, E, \textsf{inj}_i^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{inj}_i^{\mathit{con}}\ []), \mathit{exp}) \tag{137}$$

$$(\Delta, \sigma, E \circ (\textsf{inj}_i^{\mathit{con}}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \textsf{inj}_i^{\mathit{con}}\ \mathit{exp}_{\mathrm{v}}) \tag{138}$$

$$(\Delta, \sigma, E, \textsf{proj}_i^{\mathit{con}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{proj}_i^{\mathit{con}}\ []), \mathit{exp}) \tag{139}$$

$$(\Delta, \sigma, E, \textsf{tag}_{\mathit{tag}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{tag}_{\mathit{tag}}\ []), \mathit{exp}) \tag{140}$$

$$(\Delta, \sigma, E \circ (\textsf{tag}_{\mathit{tag}}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \textsf{tag}_{\mathit{tag}}\ \mathit{exp}_{\mathrm{v}}) \tag{141}$$

$$(\Delta, \sigma, E, \textsf{untag}_{\mathit{con}}^{\mathit{tag}}\ \mathit{exp}) \hookrightarrow (\Delta, \sigma, E \circ (\textsf{untag}_{\mathit{con}}^{\mathit{tag}}\ []), \mathit{exp}) \tag{142}$$

$$(\Delta, \sigma, E, \mathit{mod}.\mathit{lbl}) \hookrightarrow (\Delta, \sigma, E \circ [].\mathit{lbl}, \mathit{mod}) \tag{143}$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lbl} \triangleright \mathit{var}{=}[],\\
&\qquad \mathit{lbl}' \triangleright \mathit{var}'{=}\mathit{phrase}, \mathit{sbnds}], \mathit{val}) \hookrightarrow\\
&(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lbl} \triangleright \mathit{var}{=}\mathit{val},\\
&\qquad \mathit{lbl}' \triangleright \mathit{var}'{=}[], \{\mathit{val}/\mathit{var}\}\mathit{sbnds}], \{\mathit{val}/\mathit{var}\}\mathit{phrase})
\end{aligned}
\tag{144}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ [\mathit{sbnds}_{\mathrm{v}}, \mathit{lbl} \triangleright \mathit{var}{=}[]], \mathit{val}) \hookrightarrow\\
&\qquad (\Delta, \sigma, E, [\mathit{sbnds}_{\mathrm{v}}, \mathit{lbl} \triangleright \mathit{var}{=}\mathit{val}])
\end{aligned}
\tag{145}
$$

$$
(\Delta, \sigma, E, \mathit{mod}\ \mathit{mod}') \hookrightarrow (\Delta, \sigma, E \circ ([]\ \mathit{mod}'), \mathit{mod})
\tag{146}
$$

$$
(\Delta, \sigma, E \circ ([]\ \mathit{mod}'), \mathit{mod}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, (\mathit{mod}_{\mathrm{v}}\ []), \mathit{mod})
\tag{147}
$$

$$
(\Delta, \sigma, E, \mathit{mod}{:}\mathit{lbl}) \hookrightarrow (\Delta, \sigma, E \circ []{:}\mathit{sig}, \mathit{mod})
\tag{148}
$$

4.2.2 Reduction

$$
\begin{aligned}
&(\Delta, \sigma, E \circ (\mathit{exp_{\mathrm{v}}}^{j}\ []], \mathit{exp}_{\mathrm{v}}) \hookrightarrow\\
&\quad (\Delta, \sigma, E, \{\mathit{exp_{\mathrm{v}}}^{1}/\mathit{var}'_1\} \cdots \{\mathit{exp_{\mathrm{v}}}^{n}/\mathit{var}'_n\}\{\mathit{exp}_{\mathrm{v}}/\mathit{var}_j\}\mathit{exp}_j)\\
&\quad \text{where } \forall k \in 1..n :\\
&\quad \mathit{exp_{\mathrm{v}}}^{k} = \mathsf{fix}\,(\mathit{lbl}_i{=}(\mathit{var}'_i{:}\mathit{var}_i){\mapsto}\mathit{exp}_i)_{i=1}^{n}\ \mathsf{in}\ \mathit{var}_k\ \mathsf{end}
\end{aligned}
\tag{149}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ (\pi_{\mathit{lbl}}\ []), \{\mathit{rbnds}_{\mathrm{v}}, \mathit{lbl} \triangleright \mathit{exp}_{\mathrm{v}}, \mathit{rbnds_{\mathrm{v}}}'\}) \hookrightarrow\\
&\qquad\qquad (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}})
\end{aligned}
\tag{150}
$$

$$
(\Delta, \sigma, E \circ (\mathsf{handle}\ []\ \mathsf{with}\ \mathit{exp}), \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}})
\tag{151}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ (\mathsf{ref}^{\mathit{con}}\ []), \mathit{exp}_{\mathrm{v}}) \hookrightarrow\\
&\quad (\Delta[\mathit{loc}{:}\mathit{con}], \sigma[\mathit{loc} \mapsto \mathit{exp}_{\mathrm{v}}], E, \mathit{loc})\\
&\qquad\qquad \text{if } \mathit{loc} \notin \mathrm{BV}(\Delta)
\end{aligned}
\tag{152}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ (\mathsf{get}\ []), \mathit{loc}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}})\\
&\qquad\qquad \text{if } \sigma(\mathit{loc}) = \mathit{exp}_{\mathrm{v}}
\end{aligned}
\tag{153}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E \circ (\mathsf{set}\,(\mathit{loc}, [])), \mathit{exp}_{\mathrm{v}}) \hookrightarrow\\
&\qquad (\Delta, \sigma[\mathit{loc} \mapsto \mathit{exp}_{\mathrm{v}}], E, \{\})
\end{aligned}
\tag{154}
$$

$$
(\Delta, \sigma, E \circ (\mathsf{unroll}^{\mathit{con}}\ []), \mathsf{roll}^{\mathit{con}'}\ \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}})
\tag{155}
$$

$$
\begin{aligned}
&(\Delta, \sigma, E, \mathsf{new\_tag}[\mathit{con}]) \hookrightarrow\\
&(\Delta[\mathit{tag}{:}\mathit{con}\ \mathsf{Tag}], \sigma, E, \{\mathrm{mk} \triangleright \lambda \mathit{var}{:}\mathit{con}.\mathsf{tag}_{\mathit{tag}}\ \mathit{var},\\
&\qquad\qquad \mathrm{km} \triangleright \lambda \mathit{var}{:}\mathsf{Any}.\mathsf{untag}^{\mathit{con}}_{\mathit{tag}}\ \mathit{var}\})\\
&\qquad\qquad\qquad \text{if } \mathit{tag} \notin \mathrm{BV}(\Delta)
\end{aligned}
\tag{156}
$$

$$
(\Delta, \sigma, E \circ (\mathsf{proj}^{\mathit{con}'}_{i}\ []), \mathsf{inj}^{\mathit{con}}_{i}\ \mathit{exp}_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathit{exp}_{\mathrm{v}})
\tag{157}
$$

$$(\Delta, \sigma, E \circ (\mathsf{proj}_j^{con'}\,[\,]), \mathsf{inj}_i^{con}\ exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathsf{fail}^{con'}) \quad \text{if } i \neq j \tag{158}$$

$$(\Delta, \sigma, E \circ (\mathsf{untag}_{tag}^{con}\,[\,]), \mathsf{tag}_{tag}\ exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp_{\mathrm{v}}) \tag{159}$$

$$(\Delta, \sigma, E \circ (\mathsf{untag}_{tag'}^{con}\,[\,]), \mathsf{tag}_{tag}\ exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \mathsf{fail}^{con}) \quad \text{if } tag \neq tag' \tag{160}$$

$$(\Delta, \sigma, E \circ ((\lambda var{:}sig.mod)\,[\,]), mod_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, \{mod_{\mathrm{v}}/var\}mod) \tag{161}$$

$$(\Delta, \sigma, E \circ [\,]{:}sig, mod_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, mod_{\mathrm{v}}) \tag{162}$$

$$(\Delta, \sigma, E \circ (\mathsf{handle}\,[\,]\,\mathsf{with}\ exp') \circ RS, \mathsf{raise}^{con}\ exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, E, exp'\ exp_{\mathrm{v}}) \tag{163}$$

$$(\Delta, \sigma, RS, \mathsf{raise}^{con}\ exp_{\mathrm{v}}) \hookrightarrow (\Delta, \sigma, [\,], \mathsf{raise}^{ans}\ exp_{\mathrm{v}}) \tag{164}$$

5 External Language

5.1 Notation

As in the *Definition*, optional elements elements are enclosed by single brackets $\langle\cdots\rangle$ or double brackets $\langle\langle\cdots\rangle\rangle$. For the purposes of this grammar, all optional choices are completely independent.

5.2 Grammar of the Abstract Syntax

$$
\begin{array}{rcl}
expr & ::= & scon \\
 & \mid & longid \\
 & \mid & \{lab_1 \texttt{ = } expr_1\texttt{,} \cdots \texttt{,}lab_n \texttt{ = } expr_n\} \\
 & \mid & \texttt{let } strdec \texttt{ in } expr \texttt{ end} \\
 & \mid & expr\ expr' \\
 & \mid & expr \texttt{ : } ty \\
 & \mid & expr \texttt{ handle } match \\
 & \mid & \texttt{raise } expr \\
 & \mid & \texttt{fn } match \\
 & \mid & expr_1 \texttt{ = } expr_2 \\
 & & \\
mrule & ::= & pat \texttt{ => } expr \\
match & ::= & mrule \\
 & \mid & mrule \texttt{ | } match \\
 & & \\
strdec & ::= & \texttt{val (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } pat \texttt{ = } exp \\
 & \mid & \texttt{val (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) rec } pat \texttt{ = } exp \\
 & \mid & strdec_1\ strdec_2 \\
 & \mid & \texttt{open } longid_1 \cdots longid_n \\
 & \mid & \texttt{exception } id \\
 & \mid & \texttt{exception } id \texttt{ of } ty \\
 & \mid & \texttt{exception } id \texttt{ = } longid \\
 & \mid & \texttt{local } strdec_1 \texttt{ in } strdec_2 \texttt{ end} \\
 & \mid & \texttt{type } tybind \\
 & \mid & \texttt{datatype } datbind \\
 & \mid & \texttt{datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } tycon' \texttt{ = datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{) } longtycon \\
 & \mid & \texttt{structure } strbind \\
 & \mid & \texttt{functor } funbind
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{tybind} & ::= & \langle\texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle\langle\texttt{and}\ \mathit{tybind}\rangle\rangle \\
\mathit{datbind} & ::= & \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind} \\
 & & \langle\texttt{and}\ \mathit{datbind}\rangle \\
\mathit{conbind} & ::= & \mathit{id}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\langle\texttt{|}\ \mathit{conbind}\rangle\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{strexp} & ::= & \texttt{struct}\ \mathit{strdec}\ \texttt{end} \\
 & | & \mathit{funid}\ \texttt{(}\mathit{strexp}\texttt{)} \\
 & | & \mathit{strexp}\ \texttt{:}\ \mathit{sigexp} \\
 & | & \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} \\
 & | & \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end}
\end{array}
$$

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

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

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

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

$$
\begin{array}{rcl}
\mathit{ty} & ::= & \mathit{base} \\
 & | & \mathit{tyvar} \\
 & | & \texttt{\{}\mathit{lab}_1 \texttt{ : } \mathit{expr}_1\texttt{,} \cdots \texttt{,}\mathit{lab}_n \texttt{ : } \mathit{expr}_n\texttt{\}} \\
 & | & \langle\texttt{(}\mathit{ty}_1\texttt{,} \cdots \texttt{,}\mathit{ty}_n\texttt{)}\rangle\ \mathit{longtycon} \\
 & | & \mathit{ty}\ \texttt{->}\ \mathit{ty}' \\
 & & \\
\mathit{sigbind} & ::= & \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle\texttt{and}\ \mathit{sigbind}\rangle \\
\mathit{strbind} & ::= & \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{strbind}\rangle \\
\mathit{funbind} & ::= & \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{funbind}\rangle
\end{array}
$$

5.3 Syntactic Restrictions

- No pattern may contain the same *id* twice. No record expression, pattern, or type may contain the same *lab* twice. No *tyvar* may appear more than once in a single sequence.

- Any type variable occuring in a *conbind* must also appear in the enclosing *datbind*. Any type variable appearing in the *ty* of a `where type` must appear in the type variable sequence.

- No `val`, `type`, `datatype`, `exception`, `structure`, `signature`, or `functor` binding or specification may bind the same identifier twice; this applies also to value constructors within a *datbind*.

- In a `val rec` declaration, the pattern must be of the form

  $$\texttt{\{}\mathit{lab}_1\texttt{=}\mathit{id}_1\texttt{,} \cdots \texttt{,}\mathit{lab}_n\texttt{=}\mathit{id}_n\texttt{\}}$$

  and the expression must be of the form

  $$\texttt{\{}\mathit{lab}_1\texttt{=fn}\ \mathit{match}_1\texttt{,} \cdots \texttt{,}\mathit{lab}_n\texttt{=fn}\ \mathit{match}_n\texttt{\}.}$$

  Hence the "..." EL-notation may not appear in the pattern.

- We disallow "polymorphic recursion" in a *datbind*; any recursive use of a datatype must be applied to exactly the same type variables that explicitly scope the definition of that datatype. This disallows such code as

```
datatype 'a invalid = A | B of ('a * 'a) invalid
```

6 Elaboration

6.1 Introduction

In addition to type-checking and type-reconstruction, the elaborator performs the following tasks:

1. Datatypes are expanded into structures and signatures whose components include

   - an abstract implementation type (which is chosen to be a recursive sum type);
   - operations corresponding to constructors as values;
   - operations corresponding to constructors as patterns (deconstructors);
   - a "case" function that does branching but no destructuring.

   In particular, a datatype translates to a type and a series of constructors. The constructors are little structures containing two components; the "mk" component is the injection into the datatype while the "km" component is the projection from the datatype. Note that the mk component is always total, but the km component may fail if there is more than one constructor. (Exception constructors are handled similarly to datatype constructors, except that their "mk" and "km" components are monomorphic.)

   The "generativity" of datatypes is handled via signature ascription; the type is made opaque in the corresponding signature, and is therefore not equivalent to any other type. The matching of datatypes in signatures reduces to the matching of substructures.

2. Polymorphism (including equality polymorphism) is encoded as a use of the modules system. Polymorphic values are translated into functors, which can be explicitly instantiated with structures of types (and equality functions). More precisely, the functor takes a structure containing type constructors of kind Ω (for types which the polymorphic value requires to be equality types, the structure also contains equality functions for the instantiating types); the functor returns a structure whose single component (with label "it") is the polymorphic value made monomorphic by instantiating it with the given types.

3. Equality polymorphism, as just mentioned, and equality types are handled by explicitly constructing and passing equality functions as needed. We do not explicitly keep track of the types that admit equality; rather, a type admits equality iff the equality compiler can create an equality function for this type.

4. Patterns are expanded into uses of the appropriate record projections and datatype deconstructors. Thus the elaboration specifies a reference pattern compiler.

5. Each series of external language bindings (*strdec*) elaborates into a structure, containing a component for every variable bound in the external language. External language *identifiers* correspond to internal language *labels*.

6. Some structure labels are explicitly marked with an asterisk (lbl^*). This indicates that the structure is "open" for the purposes of identifier lookup. (See the lookup rules for more details.)

7. In structures, `open` declarations are handled by translating field names into paths. Thus there is no run-time copying/flattening of structures due to `open`.

8. All coercive aspects of the signature matching relation (the reordering and forgetting of components) are handled by introducing explicit coercion functors witnessing the relation. This makes the order and number of components of a structure apparent from its signature.

6.2 Notation

- The overbar function $\overline{\cdot}$ maps each EL identifier to an IL label. We assume that this function is injective, that the range is coinfinite in the set of IL labels, and that identifiers of different classes map to different labels. In particular, we assume that the parser distinguishes between the classes of expression variables, type constructors, type variables, structure identifiers, signature identifiers, and functor identifiers. However, we do not distinguish between an identifier being used as an expression variable, datatype constructor, or exception constructor.

  The distinguished labels "case", "eq", "it", "km", and "mk" used by the elaborator are not in the range of the overbar mapping, and other labels chosen to be fresh are similarly not in the range of the mapping.

  We extend the overbar mapping component-wise to long identifiers, which thus map to sequences of labels.

- Optional elements are enclosed in single or double angle brackets. For each rule, either all or none of the elements in single angle brackets must be present, and similarly all or none of the elements in double angle brackets must be present. Single and double angle brackets in the same rule represent two *independent* choices.

- In some cases, the optional element notation is insufficient. Therefore, we have the additional notation

  $$\left\{\begin{array}{c} element_1 \\ \text{or} \\ element_2 \end{array}\right\}$$

  which means that either $element_1$ or $element_2$ must be present. If there are multiple such choices in a single rule, this means that either the first element should always be chosen in all cases, or the second element must be chosen in all cases.

  An extension of this notation gives the choices subscripts. Then all choices with the same subscript must agree (all first element or all second element) but two choices with different subscripts are completely independent.

$$
\begin{array}{rcl}
\Gamma & ::= & \cdot \\
& | & \Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathit{con} \\
& | & \Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathit{knd}\langle = \mathit{con}\rangle \\
& | & \Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathit{sig} \\
& | & \Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathsf{Sig} = \mathit{sig}
\end{array}
$$

Figure 8: Elaboration Contexts

- The elaboration maintains an elaboration context Γ, whose grammar is shown in Figure 8. When Γ appears in an IL judgment where *decs* is expected, there is an implicit coercion which drops all top-level labels and all signature declarations.

6.3 Initial Basis

The elaborator assumes the presence of a structure $\mathit{basis}{:}\mathit{sig}_{\mathit{basis}}$ serving as the initial basis for the internal language. For our purposes here, it suffices to define the structure $\mathit{mod}_{\mathit{basis}}$ with signature $\mathit{sig}_{\mathit{basis}}$ to be the translation of

```
struct exception Bind exception Match end
```

6.4 Judgment Forms

Judgment...	*Meaning...*
$\Gamma \vdash expr \leadsto exp : con$	expression
$\Gamma \vdash match \leadsto exp : con$	pattern match
$\Gamma \vdash strdec \leadsto mod : sig$	declaration
$\Gamma \vdash strexp \leadsto mod : sig$	structure expression
$\Gamma \vdash spec \leadsto [sdecs] : \mathsf{Sig}$	signature specification
$\Gamma \vdash sigexp \leadsto sig : \mathsf{Sig}$	signature expression
$\Gamma \vdash ty \leadsto con : \Omega$	type expression
$\Gamma \vdash tybind \leadsto mod : sig$	`type` definition
$\Gamma \vdash datbind \leadsto mod : sig$	`datatype` definition
$\Gamma \vdash_{\overline{\mathrm{ctx}}} lbls \leadsto path : class$ $\Gamma \vdash_{\overline{\mathrm{ctx}}} lbls \leadsto sig : \mathsf{Sig}$	lookup in Γ
$decs, path{:}sig \vdash_{\overline{\mathrm{sig}}} lbls \leadsto lbls' : class$	lookup in signature
$sig \vdash_{\overline{\mathrm{sig}}} lbl \leadsto lbls'$	
$decs \vdash_{\overline{\mathrm{inst}}} [sdecs] \leadsto [sbnds_{\mathrm{v}}] : [sdecs']$	polymorphic instantiation
$\Gamma \vdash pat \Leftarrow exp : con \mid exp' \leadsto [sbnds] : [sbnds]$	pattern compilation
$decs \vdash_{\overline{\mathrm{eq}}} con \leadsto exp_{\mathrm{v}}$	equality compilation
$decs \vdash_{\overline{\mathrm{sub}}} sig_0 \preceq sig \leadsto \lambda var_0{:}sig_0.mod : (var_0{:}sig_0)\rightarrow sig'$	coercion compilation
$sig \vdash_{\overline{\mathrm{wt}}} lbls := con : knd \leadsto sig' : \mathsf{Sig}$	impose definition
$sig \vdash_{\overline{\mathrm{sh}}} lbls := lbls' : knd \leadsto sig' : \mathsf{Sig}$	impose sharing

6.5 Translation Rules

6.5.1 Expressions

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

$$\frac{}{\Gamma \vdash \mathit{scon} \rightsquigarrow \mathit{scon} : \mathrm{type}(\mathit{scon})} \tag{165}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : \mathit{con}}{\Gamma \vdash \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{con}} \tag{166}$$

Rule 166: This rule handles monomorphic identifiers.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : \mathit{sig} \\ \Gamma \vdash \mathit{sig} \equiv (\mathit{var}_1{:}\mathit{sig}_1){\rightarrow}\mathit{sig}'_1 : \mathsf{Sig} \\ \Gamma \vdash_{\mathrm{inst}} \mathit{sig}_{poly} \rightsquigarrow \mathit{mod}_{poly} : \mathit{sig}'_{poly} \\ \Gamma \vdash \mathit{sig} \leq \mathit{sig}'_{poly}{\rightarrow}[\mathrm{it}{\triangleright}\mathit{con}] : \mathsf{Sig}\end{array}}{\Gamma \vdash \mathit{longid} \rightsquigarrow \mathit{path}(\mathit{mod}_{poly}).\mathrm{it} : \mathit{con}} \tag{167}$$

Rule 167:

- This rule handles instantiation of polymorphic identifiers.
- Recall that all polymorphic functions are translated into total functors whose body contains a single component with the label "it".
- The module mod_{poly} is the structure of types (and equality functions) that we "guess" to instantiate the polymorphic function.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : \mathit{sig} \qquad \Gamma \vdash \mathit{path}.\mathrm{mk} : \mathit{con}}{\Gamma \vdash \mathit{longid} \rightsquigarrow \mathit{path} : \mathit{con}} \tag{168}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : \mathit{sig} \qquad \Gamma \vdash \mathit{path}.\mathrm{mk} : \mathit{sig}_{\mathrm{mk}} \\ \Gamma \vdash \mathit{sig}_{\mathrm{mk}} \equiv (\mathit{var}_1{:}\mathit{sig}_1){\rightarrow}\mathit{sig}'_1 : \mathsf{Sig} \\ \Gamma \vdash_{\mathrm{inst}} \mathit{sig}_{poly} \rightsquigarrow \mathit{mod}_{poly} : \mathit{sig}'_{poly} \\ \Gamma \vdash \mathit{sig}_{\mathrm{mk}} \leq \mathit{sig}'_{poly}{\rightarrow}[\mathrm{it}{\triangleright}\mathit{con}] : \mathsf{Sig}\end{array}}{\Gamma \vdash \mathit{longid} \rightsquigarrow ((\mathit{path}.\mathrm{mk})\mathit{mod}_{poly}).\mathrm{it} : \mathit{con}} \tag{169}$$

Rules 168 and 169: Recall that constructors translate to structures with mk and km components. When a constructor is used as a value, we translate it to the mk operation.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr}_1 \rightsquigarrow \mathit{exp}_1 : \mathit{con}_1 \ \cdots \ \Gamma \vdash \mathit{expr}_n \rightsquigarrow \mathit{exp}_n : \mathit{con}_n \\ \sigma \text{ a permutation of } 1..n \\ \mathit{var}_1, \cdots, \mathit{var}_n \notin \mathrm{FV}(\mathit{exp}_1) \cup \cdots \cup \mathrm{FV}(\mathit{exp}_n) \\ \mathit{lbl}_{\sigma(1)} < \cdots < \mathit{lbl}_{\sigma(n)}\end{array}}{\begin{array}{l}\Gamma \vdash \ \{\mathit{lab}_1 \ \texttt{=} \ \mathit{expr}_1, \cdots, \mathit{lab}_n \ \texttt{=} \ \mathit{expr}_n\} \rightsquigarrow \\ \quad [\overline{\mathit{lab}_1}{\triangleright}\mathit{var}_1{=}\mathit{exp}_1, \cdots, \overline{\mathit{lab}_n}{\triangleright}\mathit{var}_n{=}\mathit{exp}_n, \\ \quad \mathit{lbl}{\triangleright}\{\overline{\mathit{lab}_1}{\triangleright}\mathit{var}_{\sigma(1)}, \cdots, \overline{\mathit{var}_{\sigma(n)}}{\triangleright}\mathit{var}_{\sigma(n)}\}].\mathit{lbl} : \\ \quad \{\overline{\mathit{lab}_{\sigma(1)}}{\triangleright}\mathit{con}_{\sigma(1)}, \cdots, \overline{\mathit{lab}_{\sigma(n)}}{\triangleright}\mathit{con}_{\sigma(n)}\}\end{array}} \tag{170}$$

Rule 170: 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.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{mod} : \mathit{sig} \\ \mathit{var} \notin \mathrm{BV}(\Gamma) \qquad \Gamma, \mathit{lbl}^{*} \triangleright \mathit{var}{:}\mathit{sig} \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}' \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash \mathit{con}' \equiv \mathit{con} : \Omega \qquad \Gamma \vdash \mathit{con} : \Omega\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \rightsquigarrow \\ \mathsf{let}\ \mathit{var}{=}\mathit{mod}\ \mathsf{in}\ \mathit{exp}\ \mathsf{end} : \mathit{con}\end{array}} \tag{171}$$

Rule 171:

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}'' \qquad \Gamma \vdash \mathit{expr}' \rightsquigarrow \mathit{exp}' : \mathit{con}' \\ \Gamma \vdash \mathit{exp} : \mathit{con}' \rightharpoonup \mathit{con}\end{array}}{\Gamma \vdash \mathit{expr}\ \mathit{expr}' \rightsquigarrow \mathit{exp}\ \mathit{exp}' : \mathit{con}} \tag{172}$$

Rule 172: We check that the application is well-typed in the IL. (Because of our treatment of partial and total functions, it would be insufficient merely to test that $\mathit{con}'' \equiv \mathit{con}' \rightharpoonup \mathit{con}$.)

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \\ \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con}' : \Omega \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \Omega\end{array}}{\Gamma \vdash \mathit{expr}\ :\ \mathit{ty} \rightsquigarrow \mathit{exp} : \mathit{con}} \tag{173}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \\ \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{exp}' : \mathsf{Any} \rightharpoonup \mathit{con} \\ \mathit{var} \notin \mathrm{BV}(\Gamma)\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{expr}\ \texttt{handle}\ \mathit{match} \rightsquigarrow \\ \mathsf{handle}\ \mathit{exp}\ \mathsf{with} \\ \quad \lambda \mathit{var}{:}\mathsf{Any}.(\mathsf{catch}\ \mathit{exp}'\ \mathit{var}\ \mathsf{with}\ \mathsf{raise}^{\mathit{con}}\ \mathit{var}) : \\ \mathit{con}\end{array}} \tag{174}$$

Rule 174: The handling expression $\mathit{exp}'\ \mathit{var}$ may fail if the handler pattern does not match the exception raised, in which case we propagate the exception.

$$\frac{\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathsf{Any} \Gamma \vdash \mathit{con} : \Omega}{\Gamma \vdash \texttt{raise}\ \mathit{expr} \rightsquigarrow \mathsf{raise}^{\mathit{con}}\ \mathit{exp} : \mathit{con}} \tag{175}$$

Rule 175: The translation of a `raise` expression can be given any valid type *con*.

$$\frac{\Gamma \vdash match \rightsquigarrow exp : con_1 \rightharpoonup con_2 \qquad var \notin \mathrm{BV}(\Gamma)}{\begin{array}{l}\Gamma \vdash \texttt{fn}\ match \rightsquigarrow \\ \quad ((\lambda var{:}con_1.(\mathsf{catch}\ exp\ var\ \mathsf{with}\ \mathsf{raise}^{con_2}\ basis.\overline{\mathrm{Match}}.\mathrm{mk})) : \\ \quad\ con_1 \rightharpoonup con_2) : \\ \quad con_1 \rightharpoonup con_2\end{array}} \tag{176}$$

Rule 176: The expression *exp var* will fail if the match fails; here we turn the failure exception into $basis.\overline{\mathrm{Match}}.\mathrm{mk}$. We also seal the abstraction with a partial type since application of user-defined functions should not be valuable in SML '96.

$$\frac{\begin{array}{c}\Gamma \vdash expr_1 \rightsquigarrow exp_1 : con_1 \qquad \Gamma \vdash expr_2 \rightsquigarrow exp_2 : con_2 \\ \Gamma \vdash con \equiv con_1 : \Omega \qquad \Gamma \vdash con \equiv con_2 : \Omega \\ \Gamma \vdash_{\mathsf{eq}} con \rightsquigarrow exp_{\mathrm{v}}\end{array}}{\Gamma \vdash expr_1\ \texttt{=}\ expr_2 \rightsquigarrow exp_{\mathrm{v}}\,(exp_1, exp_2) : \mathsf{Bool}} \tag{177}$$

Rule 177: Translation of equality comparison; exp_{v} is the equality function, having type $con \times con \rightharpoonup \mathsf{Bool}$.

6.5.2 Matches

$$\boxed{\Gamma \vdash match \rightsquigarrow exp : con}$$

$$\frac{\begin{array}{c}var, var' \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash con' : \Omega \\ \Gamma \vdash pat \Leftarrow var' : con' \mid \mathsf{failure} \rightsquigarrow mod : sig \\ \Gamma, lbl^{*} \triangleright var{:}sig \vdash expr \rightsquigarrow exp : con\end{array}}{\begin{array}{l}\Gamma \vdash pat\ \texttt{=>}\ expr \rightsquigarrow \\ \quad \lambda var'{:}con'.\mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} : con' \rightharpoonup con\end{array}} \tag{178}$$

Rule 178:

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

$$\frac{\begin{array}{c}var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash mrule \rightsquigarrow exp : con' \rightharpoonup con \\ \Gamma \vdash match \rightsquigarrow exp' : con' \rightharpoonup con\end{array}}{\begin{array}{l}\Gamma \vdash mrule\ \texttt{|}\ match \rightsquigarrow \\ \quad \lambda var{:}con'.\mathsf{catch}\ exp\ var\ \mathsf{with}\ exp'\ var : \\ \quad con' \rightharpoonup con\end{array}} \tag{179}$$

Rule 179: The failure of pattern matching in the first clause is caught, and we try again with the next clause.

6.5.3 Declarations

$$\boxed{\Gamma \vdash \mathit{strdec} \leadsto \mathit{mod} : \mathit{sig}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con} \\ \Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{basis}.\overline{\mathrm{Bind}}.\mathrm{mk} \leadsto \mathit{mod} : \mathit{sig}\end{array}}{\Gamma \vdash \texttt{val ()}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} \leadsto \mathit{mod} : \mathit{sig}} \tag{180}$$

Rule 180: This is the monomorphic, non-recursive case.

$$\frac{\begin{array}{l}
\mathit{var}_{\mathit{poly}} \notin \mathrm{BV}(\Gamma) \qquad \mathit{lbl}', \mathit{lbl}'_1, \cdots, \mathit{lbl}'_m \notin \mathrm{rng}\overline{\cdot} \\
\mathit{sig}_{\mathit{poly}} = [\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1}^* \rhd [\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1} \rhd \mathit{var}{:}\Omega \\
\qquad\qquad\qquad\qquad \langle, \mathrm{eq} \rhd \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_1], \cdots, \\
\qquad\qquad \overline{\langle \mathit{eq}\rangle_n \mathit{tyvar}_n}^* \rhd [\overline{\langle \mathit{eq}\rangle_n \mathit{tyvar}_n} \rhd \mathit{var}{:}\Omega \\
\qquad\qquad\qquad\qquad \langle, \mathrm{eq} \rhd \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_n], \\
\qquad\qquad \mathit{lbl}'^{\,*}_1 \rhd [\mathit{lbl}'_1 \rhd \mathit{var}{:}\Omega\langle, \mathrm{eq} \rhd \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_{1'}], \cdots, \\
\qquad\qquad \mathit{lbl}'^{\,*}_m \rhd [\mathit{lbl}'_m \rhd \mathit{var}{:}\Omega\langle, \mathrm{eq} \rhd \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_{m'}]] \\
\Gamma, \mathit{lbl}^* \rhd \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}} \vdash \mathit{expr} \leadsto \mathit{exp} : \mathit{con} \\
\Gamma, \mathit{lbl}^* \rhd \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}} \vdash \mathit{pat} \\
\qquad \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{basis}.\overline{\mathrm{Bind}}.\mathrm{mk} \leadsto \mathit{mod} : \mathit{sig} \\
\Gamma, \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}} \vdash \mathit{mod} \downarrow \mathit{sig} \\
\mathit{sig} = [\mathit{lbl}_1 \rhd \mathit{con}_1, \cdots, \mathit{lbl}_k \rhd \mathit{con}_k]
\end{array}}{\begin{array}{l}
\Gamma \vdash \texttt{val}\ (\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1, \cdots, \langle \mathit{eq}\rangle_n \mathit{tyvar}_n)\ \mathit{pat}\ \texttt{=}\ \mathit{expr} \leadsto \\
\quad [\mathit{lbl}' \rhd \mathit{var}' {=} \lambda \mathit{var}{:}\mathit{sig}_{\mathit{poly}}.\mathit{mod}, \\
\quad\ \mathit{lbl}_1 \rhd \lambda \mathit{var}{:}\mathit{sig}_{\mathit{poly}}.\,[\mathrm{it} \rhd (\mathit{var}'\,(\mathit{var})).\mathit{lbl}'_1], \\
\quad\ \vdots \\
\quad\ \mathit{lbl}_k \rhd \lambda \mathit{var}{:}\mathit{sig}_{\mathit{poly}}.\,[\mathrm{it} \rhd (\mathit{var}'\,(\mathit{var})).\mathit{lbl}'_k]] : \\
\quad [\mathit{lbl}' \rhd \mathit{var}'{:}(\mathit{var}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow \mathit{sig}, \\
\quad\ \mathit{lbl}_1 \rhd (\mathit{var}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \rhd \mathit{con}_1], \\
\quad\ \vdots \\
\quad\ \mathit{lbl}_k \rhd (\mathit{var}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \rhd \mathit{con}_k]]
\end{array}} \tag{181}$$

Rule 181:

- This rule handles polymorphic, non-recursive `val` bindings.
- We assume a prepass which annotates `val` declarations with the explicit type variables implicitly scoped by that declaration. Type inference may introduce new type (or equality type) variables which not mentioned in the source (as in `val f = fn x => x` or `val f = fn x => (x=x)`).
- The translation is essentially a witness to the isomorphism $(\forall\alpha.t_1 \times t_2) \cong (\forall\alpha.t_1) \times (\forall\alpha.t_2)$.

$$
\frac{\begin{array}{c}
\mathit{var}_{\mathit{poly}} \notin \mathrm{BV}(\Gamma) \\
\mathit{sig}_{\mathit{poly}} = [\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1}^{*} \triangleright \left[\overline{\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1} \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_1\right], \cdots, \\
\overline{\langle \mathit{eq}\rangle_m \mathit{tyvar}_m}^{*} \triangleright \left[\overline{\langle \mathit{eq}\rangle_m \mathit{tyvar}_m} \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_m\right] \\
\mathit{lbl}_1'^{*} \triangleright [\mathit{lbl}_1' \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_{1'}], \cdots, \\
\mathit{lbl}_p'^{*} \triangleright \left[\mathit{lbl}_p' \triangleright \mathit{var}{:}\Omega\langle, \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool}\rangle_{p'}\right]] \\
\Gamma' = \Gamma, \mathit{lbl}^{*} \triangleright \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}}, \overline{\mathit{id}_1} \triangleright \mathit{var}_1'{:}\mathit{con}_1 \rightharpoonup \mathit{con}_1', \cdots, \overline{\mathit{id}_n} \triangleright \mathit{var}_n'{:}\mathit{con}_n \rightharpoonup \mathit{con}_n' \\
\Gamma' \vdash \mathit{match}_1 \rightsquigarrow \lambda \mathit{var}_1{:}\mathit{con}_1.\mathit{exp}_1 : \mathit{con}_1 \rightharpoonup \mathit{con}_1' \\
\vdots \\
\Gamma' \vdash \mathit{match}_n \rightsquigarrow \lambda \mathit{var}_n{:}\mathit{con}_n.\mathit{exp}_n : \mathit{con}_n \rightharpoonup \mathit{con}_n' \\
\mathit{exp}_k = \mathsf{fix}\ \mathit{var}_1'{=}(\mathit{var}_1{:}\mathit{con}_1) \mapsto \mathit{exp}_1, \cdots, \mathit{var}_m'{=}(\mathit{var}_m{:}\mathit{con}_m) \mapsto \mathit{exp}_m\ \mathsf{in}\ \mathit{var}_k'\ \mathsf{end}
\end{array}}{\begin{array}{l}
\Gamma \vdash \texttt{val}\ (\langle \mathit{eq}\rangle_1 \mathit{tyvar}_1, \cdots, \langle \mathit{eq}\rangle_n \mathit{tyvar}_n)\ \texttt{rec}\ \{(\mathit{lab}_i\texttt{=}\mathit{id}_i)_{i=1}^{n}\}\ \texttt{=}\ \{(\mathit{lab}_i\texttt{=fn}\ \mathit{match}_i)_{i=1}^{n}\} \rightsquigarrow \\
\quad [\overline{\mathit{id}_1} \triangleright \lambda \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}}.\,[\mathrm{it} \triangleright \mathit{exp}_1] \\
\quad \vdots \\
\quad \overline{\mathit{id}_n} \triangleright \lambda \mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}}.\,[\mathrm{it} \triangleright \mathit{exp}_n]] : \\
\quad [\overline{\mathit{id}_1} \triangleright (\mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \triangleright \mathit{con}_1 \rightharpoonup \mathit{con}_1'], \\
\quad \vdots \\
\quad \overline{\mathit{id}_n} \triangleright (\mathit{var}_{\mathit{poly}}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \triangleright \mathit{con}_n \rightharpoonup \mathit{con}_n']]
\end{array}} \tag{182}
$$

Rule 182: This rule handles recursive `val` bindings. As in Rule 181, we assume that the implicit scoping of explicit type variables has been made explicit by a prepass over the EL.

$$
\frac{\begin{array}{c}
\Gamma \vdash \mathit{strdec}_1 \rightsquigarrow \mathit{mod}_1 : \mathit{sig}_1 \\
\mathit{var}_1 \notin \mathrm{BV}(\Gamma) \\
\Gamma, 1^{*} \triangleright \mathit{var}_1{:}\mathit{sig}_1 \vdash \mathit{strdec}_2 \rightsquigarrow \mathit{mod}_2 : \mathit{sig}_2
\end{array}}{\begin{array}{l}
\Gamma \vdash \mathit{strdec}_1\ \mathit{strdec}_2 \rightsquigarrow \\
\qquad [1^{*} \triangleright \mathit{var}_1{=}\mathit{mod}_1, 2^{*} \triangleright \mathit{mod}_2] : \\
\qquad [1^{*} \triangleright \mathit{var}_1{:}\mathit{sig}_1, 2^{*} \triangleright \mathit{sig}_2]
\end{array}} \tag{183}
$$

Rule 183: Here we have a form of *semantic* concatenation, since syntactic concatenation could result in malformed signatures with duplicated labels (if EL identifiers are redefined). Additionally, some declarations (e.g., Rule 188) do not translate into explicit structure values.

$$
\frac{\begin{array}{c}
\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longstrid}_1} \rightsquigarrow \mathit{path}_1 : \mathit{sig}_1 \\
\vdots \\
\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longstrid}_n} \rightsquigarrow \mathit{path}_n : \mathit{sig}_n
\end{array}}{\begin{array}{l}
\Gamma \vdash \texttt{open}\ \mathit{longstrid}_1\ \cdots\ \mathit{longstrid}_n \rightsquigarrow \\
\quad [1^{*} \triangleright \mathit{path}_1, \cdots, n^{*} \triangleright \mathit{path}_n] : \\
\quad [1^{*} \triangleright \mathit{sig}_1, \cdots, n^{*} \triangleright \mathit{sig}_n]
\end{array}} \tag{184}
$$

$$\frac{\begin{array}{c}\Gamma \vdash ty \rightsquigarrow con : \Omega \\ var \notin \mathrm{BV}(\Gamma) \qquad lbl \notin \mathrm{rng}\,\overline{\cdot}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id\ \texttt{of}\ ty \rightsquigarrow \\ \quad [\overline{id}\triangleright[lbl\triangleright var{=}\mathsf{new\_tag}[con] \\ \qquad \mathrm{mk}\triangleright\pi_{\mathrm{mk}}\ var, \mathrm{km}\triangleright\pi_{\mathrm{km}}\ var] \\ \quad [\overline{id}\triangleright[lbl\triangleright var{:}\{\mathrm{mk}\triangleright con{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con\}, \\ \qquad \mathrm{mk}\triangleright con{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con]]\end{array}} \tag{185}$$

Rule 185: Note that the type of km is partial, whereas the type of mk is total.

$$\frac{var \notin \mathrm{BV}(\Gamma) \qquad lbl \notin \mathrm{rng}\,\overline{\cdot}}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id \rightsquigarrow \\ \quad \left[\overline{id}\triangleright [lbl\triangleright var{=}\mathsf{new\_tag}[\mathsf{Unit}], \mathrm{mk}\triangleright(\pi_{\mathrm{mk}}\ var)\,\{\}, \mathrm{km}\triangleright\pi_{\mathrm{km}}\ var]\right] : \\ \quad [\overline{id}\triangleright[lbl\triangleright var{:}\{\mathrm{mk}\triangleright\mathsf{Unit}{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}\mathsf{Unit}\}, \\ \qquad \mathrm{mk}\triangleright\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}\mathsf{Unit}]]\end{array}} \tag{186}$$

Rule 186: Nullary exceptions require special treatment, alas.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \\ \Gamma \vdash path.\mathrm{mk} : con_{\mathrm{mk}} \qquad \Gamma \vdash path.\mathrm{km} : \mathsf{Any}{\rightharpoonup}con\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id\ \texttt{=}\ longid \rightsquigarrow \\ \quad \left[\overline{id}\triangleright [\mathrm{mk}\triangleright path.\mathrm{mk}, \mathrm{km}\triangleright path.\mathrm{km}]\right] : \\ \quad \left[\overline{id}\triangleright [\mathrm{mk}\triangleright con_{\mathrm{mk}}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con]\right]\end{array}} \tag{187}$$

Rule 187: We check that *longid* corresponds to an exception constructor (and not a datatype constructor) by looking at the type of the km component.

$$\frac{\begin{array}{c}var_1 \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash strdec_1 \rightsquigarrow mod_1 : sig_1 \\ \Gamma, lbl^*\triangleright var_1{:}sig_1 \vdash strdec_2 \rightsquigarrow mod_2 : sig_2 \\ \Gamma, var_1{:}sig_1 \vdash sig_2 \leq sig : \mathsf{Sig} \qquad \Gamma \vdash sig : \mathsf{Sig}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \rightsquigarrow \\ \mathsf{let}\ var_1{:}mod_1\ \mathsf{in}\ mod_2\ \mathsf{end} : sig\end{array}} \tag{188}$$

Rule 188: We create a two-component structure containing all the bindings in both declarations, but do not expose the local bindings. Note that the visible bindings must be typable with respect to the ambient environment, which does not include abstract types defined locally.

$$\frac{\Gamma \vdash tybind \rightsquigarrow mod : sig}{\Gamma \vdash \texttt{type}\ tybind \rightsquigarrow mod : sig} \tag{189}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{longtycon} \leadsto path.\overline{tycon} : \Omega^n \Rightarrow \Omega \\ \Gamma \vdash path : \left[\overline{tycon} \triangleright var{:}\Omega^n \Rightarrow \Omega, lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n\right] \\ lbl_n = \text{case}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{)}\ tycon' \qquad\qquad \leadsto \\ \qquad \texttt{= datatype (}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{)}\ longtycon \\ \quad \left[\overline{tycon'}^{*} \triangleright \left[\overline{tycon'} \triangleright var{=}path.\overline{tycon}, lbl_1 \triangleright path.lbl_1, \cdots, lbl_n \triangleright path.lbl_n\right]\right] : \\ \quad \left[\overline{id'}^{*} \triangleright \left[\overline{id'} \triangleright var{:}\Omega^n \Rightarrow \Omega{=}path.\overline{tycon}, lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n\right]\right]\end{array}} \tag{190}$$

Rule 190: SML '96 adds the ability to copy datatypes. Only structures which are the translations of EL datatype declarations have a last component called "case".

$$\frac{\Gamma \vdash datbind \leadsto mod : sig}{\Gamma \vdash \texttt{datatype}\ datbind \leadsto mod : sig} \tag{191}$$

$$\frac{\Gamma \vdash strbind \leadsto mod : sig}{\Gamma \vdash \texttt{structure}\ strbind \leadsto mod : sig} \tag{192}$$

$$\frac{\Gamma \vdash funbind \leadsto mod : sig}{\Gamma \vdash \texttt{functor}\ funbind \leadsto mod : sig} \tag{193}$$

6.5.4 Structure Expressions

$$\boxed{\Gamma \vdash strexp \leadsto mod : sig}$$

$$\frac{\Gamma \vdash strdec \leadsto mod : sig}{\Gamma \vdash \texttt{struct}\ strdec\ \texttt{end} \leadsto mod : sig} \tag{194}$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{longstrid} \leadsto path : sig}{\Gamma \vdash longstrid \leadsto path : sig} \tag{195}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{funid} \leadsto path : (var_1{:}sig_1) \rightharpoonup sig_2 \\ \Gamma \vdash strexp \leadsto mod : sig \\ \Gamma \vdash_{\text{sub}} sig \preceq sig_1 \leadsto \lambda var{:}sig.mod_1 : (var{:}sig) \to sig_1' \\ \Gamma \vdash (var_1{:}sig_1) \rightharpoonup sig_2 \leq sig_1' \rightharpoonup sig_2' : \mathsf{Sig}\end{array}}{\Gamma \vdash funid\texttt{(}strexp\texttt{)} \leadsto path\,((\lambda var{:}sig.mod_1)\ mod) : sig_2'} \tag{196}$$

Rule 196:

- 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_1' is the most-specific signature of the coerced structure, which may expose more types (is a sub-signature of) sig_1.
- EL polymorphic identifiers cannot be applied as functors in the EL for two reasons: the result of doing the lookup must have a partial functor type, and the name-spaces of functor identifiers and value identifiers are disjoint.

$$\frac{\begin{array}{c} var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash strexp \leadsto mod : sig \qquad \Gamma \vdash sigexp \leadsto sig' : \mathsf{Sig} \\ \Gamma \vDash_{\mathsf{sub}} sig \preceq sig' \leadsto \lambda var{:}sig.mod' : (var{:}sig) {\to} sig'' \end{array}}{\Gamma \vdash strexp \ \texttt{:} \ sigexp \leadsto (\lambda var{:}sig.mod')\ mod : sig''} \tag{197}$$

Rule 197: As in SML, 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'' maximizes propagation of type information.

$$\frac{\begin{array}{c} \Gamma \vdash strexp \leadsto mod : sig \\ \Gamma \vdash sigexp \leadsto sig' : \mathsf{Sig} \\ var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vDash_{\mathsf{sub}} sig \preceq sig' \leadsto \lambda var{:}sig.mod' : (var{:}sig) {\to} sig'' \end{array}}{\begin{array}{l} \Gamma \vdash strexp \ \texttt{:>} \ sigexp \leadsto \\ \qquad (((\lambda var{:}sig.mod')\ mod){:}sig') : sig' \end{array}} \tag{198}$$

Rule 198: Ascribing a signature to a structure with `:>` not only hides components, but restricts information about types to that which appears in the signature.

$$\frac{\begin{array}{c} var_1 \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash strdec \leadsto mod_1 : sig_1 \\ \Gamma, lbl^{*} \triangleright var_1{:}sig_1 \vdash strexp \leadsto mod_2 : sig_2 \\ \Gamma, var_1{:}sig_1 \vdash sig_2 \leq sig : \mathsf{Sig} \qquad \Gamma \vdash sig : \mathsf{Sig} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end} \leadsto \\ \qquad \mathsf{let}\ var_1 {=} mod_1\ \mathsf{in}\ mod_2\ \mathsf{end} : sig \end{array}} \tag{199}$$

Rule 199: We create a two-component structure which includes the local module binding, but do not expose this local module. Note that the visible structure must be typable with respect to the ambient environment, which does not include abstract types in the local module.

6.5.5 Structure Bindings

$$\boxed{\Gamma \vdash strbind \leadsto mod : sig}$$

$$\frac{\Gamma \vdash strexp_1 \leadsto mod_1 : sig_1 \qquad \langle \cdots \qquad \Gamma \vdash strexp_n \leadsto mod_n : sig_n \rangle}{\begin{array}{l} \Gamma \vdash strid_1\ \texttt{=}\ strexp_1\ \langle \texttt{and}\ \cdots\ \texttt{and}\ strid_n\ \texttt{=}\ strexp_n \rangle \leadsto \\ \qquad [\overline{strid_1} \triangleright mod_1 \langle, \cdots, \overline{strid_n} \triangleright mod_n \rangle] : [\overline{strid_1} \triangleright sig_1 \langle, \cdots, \overline{strid_n} \triangleright sig_n \rangle] \end{array}} \tag{200}$$

6.5.6 Functor Bindings

$$\boxed{\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}$$

$$\frac{\begin{array}{cc} & \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{sigexp}_1 \rightsquigarrow \mathit{sig}_1 : \mathsf{Sig} & \Gamma, \overline{\mathit{sigid}_1} \triangleright \mathit{var}{:}\mathit{sig}_1 \vdash \mathit{strexp}_1 \rightsquigarrow \mathit{mod}_1 : \mathit{sig}'_1 \\ & \langle \vdots \rangle \\ \langle \Gamma \vdash \mathit{sigexp}_n \rightsquigarrow \mathit{sig}_n : \mathsf{Sig} & \Gamma, \overline{\mathit{sigid}_n} \triangleright \mathit{var}{:}\mathit{sig}_n \vdash \mathit{strexp}_n \rightsquigarrow \mathit{mod}_n : \mathit{sig}'_n \rangle \end{array}}{\begin{array}{l} \Gamma \vdash \mathit{id}_1 \texttt{(}\mathit{sigid}_1 \texttt{ : } \mathit{sigexp}_1\texttt{) = } \mathit{strexp}_1 \\ \quad \langle \texttt{and} \cdots \texttt{and } \mathit{id}_n \texttt{(}\mathit{sigid}_n \texttt{ : } \mathit{sigexp}_1\texttt{) = } \mathit{strexp}_n \rangle \rightsquigarrow \\ \quad \overline{\mathit{funid}_1} \triangleright \lambda \mathit{var}{:}\mathit{sig}_1 . \mathit{mod}_1 \langle , \cdots , \overline{\mathit{funid}_n} \triangleright \lambda \mathit{var}{:}\mathit{sig}_n . \mathit{mod}_n \rangle : \\ \quad \overline{\mathit{funid}_1} \triangleright (\mathit{var}{:}\mathit{sig}_1) \rightharpoonup \mathit{sig}'_1 \langle , \cdots , \overline{\mathit{funid}_n} \triangleright (\mathit{var}{:}\mathit{sig}_n) \rightharpoonup \mathit{sig}'_n \rangle \end{array}} \tag{201}$$

Rule 201: All functors defined explicitly in the EL have partial arrows.

6.5.7 Specifications

$$\boxed{\Gamma \vdash \mathit{spec} \rightsquigarrow [\mathit{sdecs}] : \mathsf{Sig}}$$

$$\frac{\begin{array}{c} \mathrm{FTV}(\mathit{ty}) = \emptyset \\ \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \end{array}}{\Gamma \vdash \texttt{val } \mathit{id} \texttt{ : } \mathit{ty} \rightsquigarrow [\overline{\mathit{id}} \triangleright \mathit{con}] : \mathsf{Sig}} \tag{202}$$

Rule 202: A value specifiation is monomorphic if the type contains no (free) type variables.

$$\frac{\begin{array}{c} \mathrm{FTV}(\mathit{ty}) = \{\langle \mathrm{eq} \rangle_1 \mathit{tyvar}_1, \cdots, \langle \mathrm{eq} \rangle_n \mathit{tyvar}_n\} \neq \emptyset \\ \begin{array}{rl} \mathit{sig}_{\mathit{poly}} = [\overline{\langle \mathit{eq} \rangle_1 \mathit{tyvar}_1}^* \triangleright & \left[\overline{\langle \mathit{eq} \rangle_1 \mathit{tyvar}_1} \triangleright \mathit{var}{:}\Omega \langle , \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool} \rangle_1\right], \cdots, \\ \overline{\langle \mathit{eq} \rangle_n \mathit{tyvar}_n}^* \triangleright & \left[\overline{\langle \mathit{eq} \rangle_n \mathit{tyvar}_n} \triangleright \mathit{var}{:}\Omega \langle , \mathrm{eq} \triangleright \mathit{var} \times \mathit{var} \rightharpoonup \mathsf{Bool} \rangle_n\right]] \end{array} \\ \Gamma, \mathit{lbl}^* \triangleright \mathit{var}{:}\mathit{sig}_{\mathit{poly}} \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \end{array}}{\Gamma \vdash \texttt{val } \mathit{id} \texttt{ : } \mathit{ty} \rightsquigarrow [\overline{\mathit{id}} \triangleright (\mathit{var}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \triangleright \mathit{con}]] : \mathsf{Sig}} \tag{203}$$

$$\frac{\Gamma \vdash \mathit{typdesc} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{type } \mathit{typdesc} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{204}$$

$$\frac{\Gamma \vdash \mathit{etypdesc} \rightsquigarrow [\mathit{sdecs}] : \mathsf{Sig}}{\Gamma \vdash \texttt{eqtype } \mathit{etypdesc} \rightsquigarrow [\mathit{sdecs}] : \mathsf{Sig}} \tag{205}$$

$$\frac{\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega}{\begin{array}{l} \Gamma \vdash \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \rightsquigarrow \\ \quad [\overline{\mathit{id}} \triangleright [\mathrm{mk} \triangleright \mathit{con} \rightarrow \mathsf{Any}, \mathrm{km} \triangleright \mathsf{Any} \rightharpoonup \mathit{con}]] : \mathsf{Sig} \end{array}} \tag{206}$$

$$\frac{}{\Gamma \vdash \texttt{exception } \mathit{id} \rightsquigarrow [\overline{\mathit{id}} \triangleright [\mathrm{mk} \triangleright \mathsf{Any}, \mathrm{km} \triangleright \mathsf{Any} \rightharpoonup \mathsf{Unit}]] : \mathsf{Sig}} \tag{207}$$

$$\frac{\Gamma \vdash \mathit{datbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbind} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{208}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{\mathit{longtycon}} \rightsquigarrow \mathit{path}.\overline{\mathit{tycon}} : \Omega^n \Rightarrow \Omega \\ \Gamma \vdash \mathit{path} : \left[\overline{\mathit{tycon}} \triangleright \mathit{var}{:}\Omega^n \Rightarrow \Omega, \mathit{sdecs}\right] \\ \mathit{lbl}_n = \text{case}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{datatype (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{tycon}' \qquad \rightsquigarrow \\ \quad \texttt{= datatype (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{longtycon} \\ \quad \left[\overline{\mathit{id}'}^{*} \triangleright \left[\overline{\mathit{id}'} \triangleright \mathit{var}{:}\Omega^n \Rightarrow \Omega = \mathit{path}.\overline{\mathit{tycon}}, \mathit{sdecs}\right]\right] : \mathsf{Sig}\end{array}} \tag{209}$$

$$\frac{\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{structure}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp} \rightsquigarrow \left[\overline{\mathit{strid}} \triangleright \mathit{sig}\right] : \mathsf{Sig}} \tag{210}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \\ \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma, \overline{\mathit{strid}} \triangleright \mathit{var}{:}\mathit{sig} \vdash \mathit{sigexp}' \rightsquigarrow \mathit{sig}' : \mathsf{Sig}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{functor}\ \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{:}\ \mathit{sigexp}' \rightsquigarrow \\ \quad \left[\overline{\mathit{funid}} \triangleright \mathit{var}{:}\mathit{sig} \rightharpoonup \mathit{sig}'\right] : \mathsf{Sig}\end{array}} \tag{211}$$

$$\frac{\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{include}\ \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{212}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{spec}_1 \rightsquigarrow [\mathit{sdecs}_1] : \mathsf{Sig} \qquad \Gamma, \mathit{sdecs} \vdash \mathit{spec}_2 \rightsquigarrow [\mathit{sdecs}_2] : \mathsf{Sig} \\ \Gamma \vdash [\mathit{sdecs}_1, \mathit{sdecs}_2]\ \text{ok}\end{array}}{\Gamma \vdash \mathit{spec}_1\ \mathit{spec}_2 \rightsquigarrow [\mathit{sdecs}_1, \mathit{sdecs}_2] : \mathsf{Sig}} \tag{213}$$

Rule 213: We disallow redeclaring EL identifiers in a signature. In the presence of `include`, we cannot syntactically restrict the EL to guarantee the syntactic concatenation of sdecs_1 and sdecs_2 will be well-formed—hence we check here.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \\ \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash_{\text{sig}} \overline{\mathit{longid}_1} \rightsquigarrow \mathit{lbls}_1 : \mathit{knd} \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash_{\text{sig}} \overline{\mathit{longid}_2} \rightsquigarrow \mathit{lbls}_2 : \mathit{knd} \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash \mathit{var}.\mathit{lbls}_1 \equiv \mathit{var}.\mathit{lbls}'_1 : \mathit{knd} \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash \mathit{var}.\mathit{lbls}_2 \equiv \mathit{var}.\mathit{lbls}'_2 : \mathit{knd} \\ \left\{\begin{array}{c}\mathit{sig} \vdash_{\text{sh}} \mathit{lbls}'_1 := \mathit{lbls}'_2 : \mathit{knd} \rightsquigarrow \mathit{sig}' : \mathsf{Sig} \\ \text{or} \\ \mathit{sig} \vdash_{\text{sh}} \mathit{lbls}'_2 := \mathit{lbls}'_1 : \mathit{knd} \rightsquigarrow \mathit{sig}' : \mathsf{Sig}\end{array}\right\}\end{array}}{\Gamma \vdash \mathit{spec}\ \texttt{sharing type}\ \mathit{longid}_1\ \texttt{=}\ \mathit{longid}_2 \rightsquigarrow \mathit{sig}' : \mathsf{Sig}} \tag{214}$$

Rule 214: A type component in a signature is considered abstract, and hence eligible to appear in a sharing constraint, if it is equivalent to an opaque type (type component that is not a type abbreviation) in the signature.

Here we "guess" the two opaque types to which the given components are equivalent, and patch the signature such that the opaque type with the smaller scope becomes a type abbreviation for the other opaque type.

6.5.8 Signature Expressions

$$\boxed{\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}}$$

$$\frac{\Gamma \vdash \mathit{spec} \leadsto \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{215}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{sigid}} \leadsto \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{216}$$

$$\frac{\begin{array}{c} \mathit{var}, \mathit{var}_1, \cdots, \mathit{var}_n \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma\langle, \overline{\mathit{tyvar}_1} \triangleright \mathit{var}_1{:}\Omega, \cdots, \overline{\mathit{tyvar}_n} \triangleright \mathit{var}_n{:}\Omega\rangle \vdash \mathit{ty} \leadsto \mathit{con} : \Omega \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash_{\mathrm{sig}} \overline{\mathit{longtycon}} \leadsto \mathit{lbls} : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \Gamma, \mathit{var}{:}\mathit{sig} \vdash \mathit{var}.\mathit{lbls} \equiv \mathit{var}.\mathit{lbls}' : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \mathit{sig} \vdash_{\mathrm{wt}} \mathit{lbls}' := \langle \lambda(\mathit{var}_1, \cdots, \mathit{var}_n). \rangle \mathit{con} : \langle \Omega^n \Rightarrow \rangle\, \Omega \leadsto \mathit{sig}' : \mathsf{Sig} \end{array}}{\Gamma \vdash \mathit{sigexp}\ \texttt{where type}\ \langle \texttt{(}\mathit{tyvar}_1\texttt{,}\ \cdots\ \texttt{,}\mathit{tyvar}_n\texttt{)}\rangle\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \leadsto \mathit{sig}' : \mathsf{Sig}} \tag{217}$$

Rule 217: Note that type EL type expression ty is evaluated using the ambient scope.

6.5.9 Type Expressions

$$\boxed{\Gamma \vdash ty \rightsquigarrow con : \Omega}$$

$$\frac{}{\Gamma \vdash \texttt{int} \rightsquigarrow \mathsf{Int} : \Omega} \tag{218}$$

Rule 218: There are analogous rules for the other base types (Float, Bool, Unit, String, and so on), and the base type constructors (`ref`).

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{tyvar} \rightsquigarrow path : \Omega}{\Gamma \vdash tyvar \rightsquigarrow path : \Omega} \tag{219}$$

$$\frac{\begin{array}{c}\Gamma \vdash ty_1 \rightsquigarrow con_1 : \Omega \qquad \cdots \qquad \Gamma \vdash ty_n \rightsquigarrow con_n : \Omega \\ \sigma \text{ a permutation of } 1..n \\ lab_{\sigma(1)} < \cdots < lab_{\sigma(n)}\end{array}}{\Gamma \vdash \texttt{\{}lab_1 \texttt{:} ty_1, \cdots, lab_n \texttt{:} ty_n\texttt{\}} \rightsquigarrow \{\overline{lab_{\sigma(1)}} \triangleright con_{\sigma(1)}, \cdots, \overline{lab_{\sigma(n)}} \triangleright con_{\sigma(n)}\} : \Omega} \tag{220}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{longtycon} \rightsquigarrow path : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \langle \Gamma \vdash ty_1 \rightsquigarrow con_1 : \Omega \quad \cdots \quad \Gamma \vdash ty_n \rightsquigarrow con_n : \Omega \rangle\end{array}}{\Gamma \vdash \langle \texttt{(}ty_1\texttt{,} \cdots \texttt{,}ty_n\texttt{)} \rangle\ longtycon \rightsquigarrow path\langle (con_1, \cdots, con_n) \rangle : \Omega} \tag{221}$$

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \Omega \qquad \Gamma \vdash ty' \rightsquigarrow con' : \Omega}{\Gamma \vdash ty\ \texttt{->}\ ty' \rightsquigarrow con \rightharpoonup con' : \Omega} \tag{222}$$

Rule 222: There is no way to directly denote a total ($\rightarrow$) type in the external language.

6.5.10 Type Definitions

$$\boxed{\Gamma \vdash tybind \rightsquigarrow [sbnds] : [sdecs]}$$

$$\frac{\begin{array}{c}\langle var_1, \cdots, var_n \notin \mathrm{BV}(\Gamma) \rangle \\ \Gamma, \langle \overline{tyvar_1} \triangleright var_1 {:} \Omega, \cdots, \overline{tyvar_n} \triangleright var_n {:} \Omega \rangle \vdash ty \rightsquigarrow con' : \Omega \\ con = \langle \lambda(var_1, \cdots, var_n). \rangle con' \\ \langle\langle \Gamma \vdash tybind \rightsquigarrow [sbnds] : [sdecs] \qquad var \notin \mathrm{FV}(sdecs) \rangle\rangle\end{array}}{\begin{array}{l}\Gamma \vdash \langle \texttt{(}tyvar_1\texttt{,} \cdots \texttt{,}tyvar_n\texttt{)} \rangle\ tycon\ \texttt{=}\ ty\ \langle\langle \texttt{and}\ tybind \rangle\rangle \rightsquigarrow \\ \qquad [\overline{tycon} \triangleright var = con \langle\langle, sbnds \rangle\rangle] : \\ \qquad [\overline{tycon} \triangleright var {:} \langle \Omega^n \Rightarrow \rangle\, \Omega = con \langle\langle, sdecs \rangle\rangle]\end{array}} \tag{223}$$

6.5.11 Type Descriptions

$$\boxed{\Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig}}$$

$$\frac{\langle\langle \Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig} \rangle\rangle}{\begin{array}{l}\Gamma \vdash \texttt{type}\ \langle \texttt{(}tyvar_1, \cdots, tyvar_n\texttt{)} \rangle\ id\ \langle\langle\ \texttt{and}\ typdesc \rangle\rangle \rightsquigarrow \\ \qquad [\overline{id} \triangleright \langle \Omega^n \Rightarrow \rangle\, \Omega \langle\langle, sdecs \rangle\rangle] : \mathsf{Sig}\end{array}} \tag{224}$$

$$\frac{\begin{array}{c}\langle var_1, \cdots, var_n \notin \mathrm{BV}(\Gamma)\rangle \\ \Gamma, \langle \overline{tyvar_1} \triangleright var_1{:}\Omega, \cdots, \overline{tyvar_n} \triangleright var_n{:}\Omega\rangle \vdash ty \rightsquigarrow con' : \Omega \\ con := \langle \lambda(var_1, \cdots, var_n).\rangle con' \\ \langle\langle \Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig}\rangle\rangle \qquad \langle\langle var \notin \mathrm{FV} sdecs\rangle\rangle\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{type}\ \langle \texttt{(}tyvar_1, \cdots, tyvar_n\texttt{)}\rangle\ id\ \texttt{=}\ ty\ \langle\langle\ \texttt{and}\ typdesc\rangle\rangle \rightsquigarrow \\ \quad \left[\overline{id} \triangleright \langle \Omega^n \Rightarrow\rangle\, \Omega{=}con\langle\langle, sdecs\rangle\rangle\right] : \mathsf{Sig}\end{array}} \tag{225}$$

6.5.12 Equality Type Descriptions

$$\boxed{\Gamma \vdash etypdesc \rightsquigarrow [sdecs] : \mathsf{Sig}}$$

The polymorphic equality functions defined by this judgment the same as the default "structural" equality supplied by SML. Essentially, a type is an equality type if and only if we can use this judgment to create the equality function at that type.

$$\frac{\langle \Gamma \vdash etypdesc \rightsquigarrow [sdecs] : \mathsf{Sig}\rangle}{\begin{array}{l}\Gamma \vdash id\ \langle \texttt{and}\ etypdesc\rangle \rightsquigarrow \\ \quad \left[\overline{id}^{*} \triangleright \left[\overline{id} \triangleright var{:}\Omega, \mathrm{eq} \triangleright var \times var \rightharpoonup \mathsf{Bool}\right] \langle, sdecs\rangle\right] : \mathsf{Sig}\end{array}} \tag{226}$$

$$\frac{\begin{array}{c} sig' = [\overline{tyvar_1}^{*} \triangleright \left[\overline{tyvar_1} \triangleright var_1{:}\Omega, \mathrm{eq} \triangleright var_1 \times var_1 \rightharpoonup \mathsf{Bool}\right], \cdots, \\ \overline{tyvar_n}^{*} \triangleright \left[\overline{tyvar_n} \triangleright var_n{:}\Omega, \mathrm{eq} \triangleright var_n \times var_n \rightharpoonup \mathsf{Bool}\right]] \\ con = var(var'.\overline{tyvar_1}^{*}.\overline{tyvar_1}, \cdots, var'.\overline{tyvar_n}^{*}.\overline{tyvar_n}) \\ \langle \Gamma \vdash etypdesc \rightsquigarrow [sdecs] : \mathsf{Sig}\rangle \end{array}}{\begin{array}{l}\Gamma \vdash (tyvar_1, \cdots, tyvar_n)\ id\ \langle \texttt{and}\ etypdesc\rangle \rightsquigarrow \\ \quad \left[\overline{id}^{*} \triangleright \left[\overline{id} \triangleright var{:}\Omega^n \Rightarrow \Omega, \mathrm{eq} \triangleright (var'{:}sig') \rightarrow [\mathrm{it} \triangleright con \times con \rightharpoonup \mathsf{Bool}]\right] \langle, sdecs\rangle\right] : \\ \quad \mathsf{Sig}\end{array}} \tag{227}$$

6.5.13 Datatype Definitions

$$\boxed{\Gamma \vdash \mathit{datbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}$$

To make the definition somewhat manageable, we define the necessary many types and expressions in steps:

- The mappings $\sigma_1, \cdots, \sigma_p$ are permutations of the indices $1..k$. We use these to require that all datatypes being defined simultaneously be parametrized by the same set of type variables. This can always be achieved by breaking datatype bindings into strongly-connected components.
- $tyvar_1, \cdots, tyvar_k$ are the EL type variables scoped in each datatype definition, which correspond to the IL type variables $var_1^{poly}, \cdots, var_k^{poly}$. In functors created for polymorphism, these correspond to structure projections $var^{poly}.\overline{tyvar_1}, \cdots, var^{poly}.\overline{tyvar_k}$.
- $tycon_1, \cdots, tycon_p$ are the "external" EL names of the type constructors being defined. Essentially, given values for the type variables, we define the datatypes instantiated at these values; the "internal" EL names are $id'_1, \cdots, id'_p$, and the IL names are $var_1^{dt}, \cdots, var_p^{dt}$.
- In the following names, it may help to remember the following association: C="in Constructor" (i.e., in terms of $var_1^{poly}, \cdots, var_k^{poly}$), F="in Functor" (i.e., in terms of $var^{poly}.\overline{tyvar_1}, \cdots, var^{poly}.\overline{tyvar_k}$), R="recursive" (i.e., contains $var_1^{dt}, \cdots, var_p^{dt}$ free), NR="non-recursive".
- $con^{\mathrm{NR/C}}$ is a constructor tuple of the datatypes, with the variables $var_1^{poly}, \cdots, var_k^{poly}$, (and variables corresponding to the datatypes at these types, $var_1^{dt}, \cdots, var_p^{dt}$) free.
- For $i \in 1..p$, $con_i^{\mathrm{R/C}}$ is the i^{th} datatype parametrized by the type variables $var_1^{poly}, \cdots, var_k^{poly}$.
- In the translation, for $i \in 1..p$ we bind var_i to be the type function that takes the var_j^{poly}'s in the correct order, and returns $con_i^{\mathrm{R/C}}$ instantated there. These are the implementation of the datatypes, and these definitions will be hidden by signature ascription to make the datatypes abstract.
- Therefore for $i \in 1..p$, we have $con_i^{\mathrm{R/F}}$ as the i^{th} type constructor instantiated at the types $var^{poly}.\overline{tyvar_1}, \cdots, var^{poly}.\overline{tyvar_1}$.
- For $i \in 1..p$ and $j \in 1..m_p$, $con_{ij}^{\mathrm{R/F}}$ is the translation of ty_{ij} (if this constructor carries a type; Unit otherwise); these are parametrized by the type projections $var^{poly}.\overline{tyvar_1}, \cdots,$ $var^{poly}.\overline{tyvar_1}$.

NOTE: For space and readability reasons, the following rule shows only the module to which the datatype declaration elaborates and elides the signature of the module. However since the module is a sealed structure, its signature is exactly the same as the sealing signature.

$\sigma_1, \cdots, \sigma_p$ permutations of $1..k$

$$\Gamma' := \Gamma, \overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega, \cdots, \overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega, \overline{tycon_1} \triangleright var_1^{dt}{:}\Omega \cdots, \overline{tycon_p} \triangleright var_p^{dt}{:}\Omega$$

$$\Gamma' \vdash \left\{ \begin{matrix} \texttt{unit} \\ \text{or} \\ ty_{ij} \end{matrix} \right\}_{ij} \leadsto con_{ij}^{\mathrm{NR/C}} : \Omega$$

$$con^{\mathrm{NR/C}} := \left(\Sigma \left(con_{11}^{\mathrm{NR/C}}, \cdots, con_{1m_1}^{\mathrm{NR/C}} \right), \cdots, \Sigma \left(con_{p1}^{\mathrm{NR/C}}, \cdots, con_{pm_p}^{\mathrm{NR/C}} \right) \right)$$

$$con_i^{\mathrm{R/C}} := \mu_i \left(\lambda(var_1^{dt}, \cdots, var_p^{dt}).con^{\mathrm{NR/C}} \right)$$

$$con_i^{\mathrm{R/F}} := var \left(var^{poly}.\overline{tyvar_1}, \cdots, var^{poly}.\overline{tyvar_k} \right)$$

$$\phi^{\mathrm{C}\to\mathrm{F}} := \{ var^{poly}.\overline{tyvar_1} / var_1^{poly} \} \cdots \{ var^{poly}.\overline{tyvar_k} / var_k^{poly} \}$$

$$\phi^{\mathrm{NR}\to\mathrm{R}} := \{ con_1^{\mathrm{R/F}} / var_1^{dt} \} \cdots \{ con_k^{\mathrm{R/F}} / var_k^{dt} \}$$

$$con_{ij}^{\mathrm{R/F}} := \phi^{\mathrm{C}\to\mathrm{F}} \phi^{\mathrm{NR}\to\mathrm{R}} con_{ij}^{\mathrm{NR/C}}$$

$$sig^{poly} := \left[\overline{tyvar_1}^* \triangleright \left[\overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega \right], \cdots, \overline{tyvar_k}^* \triangleright \left[\overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega \right] \right]$$

$$mod_{ij}^{\mathrm{mk}} := \lambda var^{poly}{:}sig^{poly}. \left[\mathrm{it} \triangleright \left\{ \begin{matrix} \mathsf{roll}^{con_i^{\mathrm{R/F}}} \left(\mathsf{inj}_j^{\Sigma\left(con_{i1}^{\mathrm{R/F}}, \cdots, con_{im_i}^{\mathrm{R/F}}\right)} \{\} \right) \\ \text{or} \\ \lambda var{:}con_{ij}^{\mathrm{R/F}}.\mathsf{roll}^{con_i^{\mathrm{R/F}}} \left(\mathsf{inj}_j^{\Sigma\left(con_{i1}^{\mathrm{R/F}}, \cdots, con_{im_i}^{\mathrm{R/F}}\right)} var \right) \end{matrix} \right\}_{ij} \right]$$

$$sig_{ij}^{\mathrm{mk}} := var^{poly}{:}sig^{poly} \rightharpoonup \left[\mathrm{it} \triangleright \left\{ \begin{matrix} con_i^{\mathrm{R/F}} \\ \text{or} \\ con_{ij}^{\mathrm{R/F}} \to con_i^{\mathrm{R/F}} \end{matrix} \right\}_{ij} \right]$$

$$mod_{ij}^{\mathrm{km}} := \lambda var^{poly}{:}sig^{poly}. \left[\mathrm{it} \triangleright \lambda var{:}con_i^{\mathrm{R/F}}.\mathsf{proj}_j^{con_{ij}^{\mathrm{R/F}}} \left(\mathsf{unroll}^{con_i^{\mathrm{R/F}}} var \right) \right]$$

$$sig_{ij}^{\mathrm{km}} := var^{poly}{:}sig^{poly} \rightharpoonup \left[\mathrm{it} \triangleright \begin{cases} con_i^{\mathrm{R/F}} \to con_{ij}^{\mathrm{R/F}} & \text{if } m_i = 1; \\ con_i^{\mathrm{R/F}} \rightharpoonup con_{ij}^{\mathrm{R/F}} & \text{otherwise.} \end{cases} \right]$$

$$sig_{poly}^{+} := [\overline{tyvar_1}^* \triangleright \left[\overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega \right], \cdots, \overline{tyvar_k}^* \triangleright \left[\overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega \right], lbl^* \triangleright [lbl \triangleright \Omega]]$$

$$mod_i^{\mathrm{case}} := \lambda var^{poly}{:}sig^{poly}.[\mathrm{it} \triangleright \lambda var{:}\{1 \triangleright con_i^{\mathrm{R/F}} \rightharpoonup var_{poly}.lbl, \cdots, m_i \triangleright con_i^{\mathrm{R/F}} \rightharpoonup var_{poly}.lbl\}.\lambda var'{:}con_i^{\mathrm{R/F}}.$$
$$\mathsf{case}^{\Sigma\left(con_{i1}^{\mathrm{R/F}}, \cdots, con_{im_i}^{\mathrm{R/F}}\right)} \mathsf{unroll}^{con_i^{\mathrm{R/F}}} var' \ \mathsf{of}\ (\pi_{\overline{1}}\, var) var', \cdots, (\pi_{\overline{m_i}}\, var) var' \ \mathsf{end}]$$

$$sig_i^{\mathrm{case}} := var^{poly}{:}sig^{poly} \rightharpoonup \left[\mathrm{it} \triangleright \{1 \triangleright con_i^{\mathrm{R/F}} \rightharpoonup var_{poly}.lbl, \cdots, m_i \triangleright con_i^{\mathrm{R/F}} \rightharpoonup var_{poly}.lbl\} \to con_i^{\mathrm{R/F}} \rightharpoonup var_{poly}.lbl \right]$$

$$sig_{poly}^{=} := [\overline{tyvar_1}^* \triangleright \left[\overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega, \mathrm{eq} \triangleright var_1^{poly} \times var_1^{poly} \rightharpoonup \mathsf{Bool} \right], \cdots$$
$$\overline{tyvar_k}^* \triangleright \left[\overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega, \mathrm{eq} \triangleright var_k^{poly} \times var_k^{poly} \rightharpoonup \mathsf{Bool} \right]]$$

$$\langle \Gamma, {lbl_{poly}^{=}}^* \triangleright var_{poly}^{=}{:}sig_{poly}^{=} \vdash_{\mathrm{eq}} con_i^{\mathrm{R/F}} \leadsto exp_{\mathrm{v}_i}^{\mathrm{eq}} \rangle_{i'}$$

$$\langle mod_i^{\mathrm{eq}} := \lambda var^{poly}{:}sig^{poly}.\, [\mathrm{it} \triangleright exp_{\mathrm{v}_i}^{\mathrm{eq}}] \rangle_{i'}$$

$$\langle sig_i^{\mathrm{eq}} := var^{poly}{:}sig^{poly} \rightharpoonup \left[\mathrm{it} \triangleright con_i^{\mathrm{R/F}} \times con_i^{\mathrm{R/F}} \rightharpoonup \mathsf{Bool} \right] \rangle_{i'}$$

$$
\begin{aligned}
&\Gamma \vdash (\mathit{tyvar}_{\sigma_1(1)},\cdots,\mathit{tyvar}_{\sigma_1(k)})\ \mathit{tycon}_1\texttt{=}\mathit{id}'_1\ \texttt{with}\ \mathit{id}_{11}\left\{\begin{matrix}\text{or}\\ \texttt{of}\ \mathit{ty}_{11}\end{matrix}\right\}_{11}\texttt{|}\cdots\texttt{|}\,\mathit{id}_{1m_1}\left\{\begin{matrix}\text{or}\\ \texttt{of}\ \mathit{ty}_{1m_1}\end{matrix}\right\}_{1m_1}\\
&\quad \texttt{and}\ \cdots\texttt{and}\\
&\quad (\mathit{tyvar}_{\sigma_p(1)},\cdots,\mathit{tyvar}_{\sigma_p(k)})\ \mathit{tycon}_p\texttt{=}\mathit{id}'_p\ \texttt{with}\ \mathit{id}_{p1}\left\{\begin{matrix}\text{or}\\ \texttt{of}\ \mathit{ty}_{p1}\end{matrix}\right\}_{p1}\texttt{|}\cdots\texttt{|}\,\mathit{id}_{pm_p}\left\{\begin{matrix}\text{or}\\ \texttt{of}\ \mathit{ty}_{pm_p}\end{matrix}\right\}_{pm_p}\leadsto\\
&\quad\Big[\mathit{lbl}_1\triangleright \mathit{var}_1{=}\lambda(\mathit{var}^{\mathit{poly}}_{\sigma_1(1)},\cdots,\mathit{var}^{\mathit{poly}}_{\sigma_1(k)}).\mathit{con}_1^{\mathrm{R/C}},\cdots,\\
&\quad\ \mathit{lbl}_p\triangleright \mathit{var}_p{=}\lambda(\mathit{var}^{\mathit{poly}}_{\sigma_p(1)},\cdots,\mathit{var}^{\mathit{poly}}_{\sigma_p(k)}).\mathit{con}_p^{\mathrm{R/C}},\\
&\quad\ \overline{\mathit{tycon}_1}^{*}\triangleright[\overline{\mathit{tycon}_1}\triangleright \mathit{var}_1,\\
&\qquad\qquad \overline{\langle \mathrm{eq}\triangleright \mathit{mod}_1^{\mathrm{eq}},\rangle_{1'}}\\
&\qquad\qquad \overline{\mathit{id}_{11}}\triangleright\left[\mathrm{mk}\triangleright \mathit{mod}^{\mathrm{mk}}_{11},\mathrm{km}\triangleright \mathit{mod}^{\mathrm{km}}_{11}\right],\\
&\qquad\qquad \vdots\\
&\qquad\qquad \overline{\mathit{id}_{1m_1}}\triangleright\left[\mathrm{mk}\triangleright \mathit{mod}^{\mathrm{mk}}_{1m_1},\mathrm{km}\triangleright \mathit{mod}^{\mathrm{km}}_{1m_1}\right],\\
&\qquad\qquad \mathrm{case}\triangleright \mathit{mod}_1^{\mathrm{case}}],\\
&\quad\ \vdots\\
&\quad\ \overline{\mathit{tycon}_p}^{*}\triangleright[\overline{\mathit{tycon}_p}\triangleright \mathit{var}_p,\\
&\qquad\qquad \overline{\langle \mathrm{eq}\triangleright \mathit{mod}_p^{\mathrm{eq}},\rangle_{p'}}\\
&\qquad\qquad \overline{\mathit{id}_{p1}}\triangleright\left[\mathrm{mk}\triangleright \mathit{mod}^{\mathrm{mk}}_{p1},\mathrm{km}\triangleright \mathit{mod}^{\mathrm{km}}_{p1}\right],\\
&\qquad\qquad \vdots\\
&\qquad\qquad \overline{\mathit{id}_{pm_p}}\triangleright\left[\mathrm{mk}\triangleright \mathit{mod}^{\mathrm{mk}}_{pm_p},\mathrm{km}\triangleright \mathit{mod}^{\mathrm{km}}_{pm_p}\right],\\
&\qquad\qquad \mathrm{case}\triangleright \mathit{mod}_p^{\mathrm{case}}]\Big]\ :\\
&\quad[\mathit{lbl}_1\triangleright \mathit{var}_1{:}\Omega^k{\Rightarrow}\Omega,\cdots,\mathit{lbl}_p\triangleright \mathit{var}_p{:}\Omega^k{\Rightarrow}\Omega,\\
&\quad\ \overline{\mathit{tycon}_1}^{*}\triangleright[\overline{\mathit{tycon}_1}\triangleright\Omega^k{\Rightarrow}\Omega{=}\mathit{var}_1,\\
&\qquad\qquad \overline{\langle \mathrm{eq}\triangleright \mathit{sig}_1^{\mathrm{eq}},\rangle_{1'}}\\
&\qquad\qquad \overline{\mathit{id}_{11}}\triangleright\left[\mathrm{mk}\triangleright \mathit{sig}^{\mathrm{mk}}_{11},\mathrm{km}\triangleright \mathit{sig}^{\mathrm{km}}_{11}\right],\\
&\qquad\qquad \vdots\\
&\qquad\qquad \overline{\mathit{id}_{1m_1}}\triangleright\left[\mathrm{mk}\triangleright \mathit{sig}^{\mathrm{mk}}_{1m_1},\mathrm{km}\triangleright \mathit{sig}^{\mathrm{km}}_{1m_1}\right],\\
&\qquad\qquad \mathrm{case}\triangleright \mathit{sig}_1^{\mathrm{case}}],\\
&\quad\ \vdots\\
&\quad\ \overline{\mathit{tycon}_p}^{*}\triangleright[\overline{\mathit{tycon}_p}\triangleright\Omega^k{\Rightarrow}\Omega{=}\mathit{var}_p,\\
&\qquad\qquad \overline{\langle \mathrm{eq}\triangleright \mathit{sig}_p^{\mathrm{eq}},\rangle_{p'}}\\
&\qquad\qquad \overline{\mathit{id}_{p1}}\triangleright\left[\mathrm{mk}\triangleright \mathit{sig}^{\mathrm{mk}}_{p1},\mathrm{km}\triangleright \mathit{sig}^{\mathrm{km}}_{p1}\right],\\
&\qquad\qquad \vdots\\
&\qquad\qquad \overline{\mathit{id}_{pm_p}}\triangleright\left[\mathrm{mk}\triangleright \mathit{sig}^{\mathrm{mk}}_{pm_p},\mathrm{km}\triangleright \mathit{sig}^{\mathrm{km}}_{pm_p}\right],\\
&\qquad\qquad \mathrm{case}\triangleright \mathit{sig}_p^{\mathrm{case}}]\Big]
\end{aligned}
\tag{228}
$$

6.6 Polymorphic Instantiation

$$\boxed{decs \vdash_{\text{inst}} [sdecs] \rightsquigarrow [sbnds_{\text{v}}] : [sdecs']}$$

$$\frac{\begin{array}{c} decs \vdash con : \Omega \\ \langle decs \vdash_{\text{eq}} con \rightsquigarrow exp_{\text{v}} \rangle \\ decs \vdash_{\text{inst}} [sdecs] \rightsquigarrow [sbnds_{\text{v}}] : [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\text{inst}} [lbl' \triangleright [lbl \triangleright var{:}\Omega \langle , \text{eq} \triangleright var \times var \rightharpoonup \mathsf{Bool} \rangle] \, , sdecs] \rightsquigarrow \\ \quad [lbl' \triangleright [lbl \triangleright var{=}con \langle , \text{eq} \triangleright exp_{\text{v}} \rangle] \, , sbnds_{\text{v}}] : \\ \quad [lbl' \triangleright [lbl \triangleright var{:}\Omega{=}con \langle , \text{eq} \triangleright var \times var \rightharpoonup \mathsf{Bool} \rangle] \, , sdecs'] \end{array}} \tag{229}$$

Rule 229 Given the domain signature for the functor corresponding to a polymorphic function, nondeterministically choose types and equality functions so as to match the signature.

6.7 Equality Compilation

$$\boxed{decs \vdash_{\text{eq}} con \rightsquigarrow exp_{\text{v}}}$$

$$\frac{}{decs \vdash_{\text{eq}} base \rightsquigarrow =_{base}} \tag{230}$$

Rule 230: We assume primitive equality operations already exist at base types such as Int and Float.

$$\frac{\begin{array}{c} decs \vdash_{\text{eq}} con_1 \rightsquigarrow exp_{\text{v}1} \quad \cdots \quad decs \vdash_{\text{eq}} con_n \rightsquigarrow exp_{\text{v}n} \\ decs \vdash con \equiv \{lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n\} : \Omega \end{array}}{\begin{array}{l} decs \vdash_{\text{eq}} con \rightsquigarrow \\ \quad \lambda var{:}con \times con.exp_{\text{v}1}(\pi_{lbl_1}(\pi_{\overline{1}}\, var), \pi_{lbl_1}(\pi_{\overline{2}}\, var)) \text{ and} \\ \qquad \cdots \text{ and } exp_{\text{v}n}(\pi_{lbl_n}(\pi_{\overline{1}}\, var), \pi_{lbl_n}(\pi_{\overline{2}}\, var)) \end{array}} \tag{231}$$

Rule 231: Records are compared component-wise for equality.

$$\frac{\begin{array}{l} decs \vdash con \equiv \Sigma\,(con_1, \cdots, con_n) : \Omega \\ decs \vdash_{\text{eq}} con_1 \rightsquigarrow exp_{\text{v}1} \quad \cdots \quad decs \vdash_{\text{eq}} con_n \rightsquigarrow exp_{\text{v}n} \\ \forall i \in 1..n : \\ \quad exp'_{\text{v}i} = \lambda var'{:}con.exp_{\text{v}i}\,(\mathsf{proj}_i^{con_i}(\pi_{\overline{1}}\, var), \mathsf{proj}_i^{con_i}(\pi_{\overline{2}}\, var)) \end{array}}{\begin{array}{l} decs \vdash_{\text{eq}} con \rightsquigarrow \\ \quad \lambda var{:}con \times con.\mathsf{catch\ case}^{con}\,(\pi_{\overline{1}}\, var) \text{ of } exp'_{\text{v}1}, \cdots, exp'_{\text{v}n} \text{ end} \\ \qquad \mathsf{with\ false} \end{array}} \tag{232}$$

Rule 232: Values of a sum type are equal if they have the same tag and the tagged values are equal.

$$
\frac{\begin{array}{c} decs \vdash con \equiv \mu_j\ con' : \Omega \\ decs \vdash con' : \Omega^n \Rightarrow \Omega^n \\ decs, (var'_i\!: [lbl_i \triangleright var_i\!:\!\Omega, \mathrm{eq} \triangleright var_i \times var_i \rightharpoonup \mathsf{Bool}])_{i=1}^n \vdash_{\mathrm{eq}} \\ con'\,(var'_1.lbl_1, \cdots, var'_n.lbl_n) \leadsto exp_{\mathrm{v}\,i} \\ \forall i \in 1..n : \\ exp_i^{\mathrm{eq}} = (\{var_1^{\mathrm{eq}}/var'_1.\mathrm{eq}\} \cdots \{var_n^{\mathrm{eq}}/var'_n.\mathrm{eq}\}\, exp_{\mathrm{v}\,i}) \\ (\mathsf{unroll}^{\mu_i\ con'}\,(\pi_{\overline{1}}\,var), \mathsf{unroll}^{\mu_i\ con'}\,(\pi_{\overline{2}}\,var)) \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{eq}} con \leadsto \\ \quad \mathsf{fix}\,(var_i^{\mathrm{eq}} = (var{:}\mu_i\ con' \times \mu_i\ con') \mapsto exp_i^{\mathrm{eq}})_{i=1}^n \\ \quad \mathsf{in}\ var_j^{\mathrm{eq}}\ \mathsf{end} \end{array}} \tag{233}
$$

Rule 233: A recursive type generates recursively-defined equality functions.

$$
\frac{decs \vdash path.\mathrm{eq} : path.lbl \times path.lbl \rightharpoonup \mathsf{Bool}}{decs \vdash_{\mathrm{eq}} path.lbl \leadsto path.\mathrm{eq}} \tag{234}
$$

Rule 234: If the constructor is simply a path—the name of an abstract type—we look for an equality function for that type in the same structure.

$$
\frac{\begin{array}{c} decs \vdash con \equiv path.lbl\,(con_1, \cdots, con_n) : \Omega \\ decs \vdash path.\mathrm{eq} : sig_{\mathrm{eq}} \\ decs \vdash sig_{\mathrm{eq}} \equiv (var{:}sig) \rightarrow [\mathrm{it} \triangleright con_{\mathrm{it}}] : \mathsf{Sig} \\ decs \vdash_{\mathrm{inst}} sig \leadsto mod_{\mathrm{v}} : sig' \\ decs \vdash sig_{\mathrm{eq}} \leq sig' \rightarrow [\mathrm{it} \triangleright con] : \mathsf{Sig} \end{array}}{decs \vdash_{\mathrm{eq}} con \leadsto (path.\mathrm{eq}\,(mod_{\mathrm{v}})).\mathrm{it}} \tag{235}
$$

Rule 235: Same as Rule 234, except here we deal with the application of an abstract type to a tuple of types. The instantiation judgment may recursively invoke the equality compiler, in order to create equality functions for $con_1, \cdots, con_n$.

6.8 Pattern Compilation

Patterns

$$\boxed{\Gamma \vdash pat \Leftarrow exp : con \mid exp' \leadsto [sdecs] : [sbnds]}$$

The judgment should be read as "the bindings of the result of matching *exp* to the pattern *pat* are [*sdecs*] : [*sbnds*]; if the pattern match fails for any reason, then the exception *exp'* is raised." In practice, *exp'* is either failure or *basis*.$\overline{\text{Bind}}$.mk.

$$\frac{\Gamma \vdash_{\mathsf{ctx}} \overline{id} \not\leadsto \quad \text{or} \quad \Gamma \vdash_{\mathsf{ctx}} \overline{id} \leadsto path : con' \qquad var \notin \mathrm{BV}(\Gamma)}{\Gamma \vdash id \Leftarrow exp : con \mid exp' \leadsto [\overline{id} \triangleright var{=}exp] : [\overline{id} \triangleright var{:}con]} \tag{236}$$

Rule 236: Pattern match against an identifier that is not a constructor. (The premise is equivalent to "rules 241 and 243 do not apply".)

$$\frac{lbl \text{ fresh} \qquad var \notin \mathrm{BV}(\Gamma) \qquad \mathrm{type}(scon) = con \qquad \Gamma \vdash exp' : \mathsf{Unit}}{\Gamma \vdash scon \Leftarrow exp : con \mid exp' \leadsto [lbl \triangleright var{=}\mathsf{if}^{\mathsf{Unit}}\ exp{=}_{con} scon\ \mathsf{then}\ \{\}\ \mathsf{else\ raise}^{\mathsf{Unit}}\ exp'] : [lbl \triangleright var{:}\mathsf{Unit}]} \tag{237}$$

Rule 237: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{lbl \text{ fresh} \qquad var \notin \mathrm{BV}(\Gamma)}{\Gamma \vdash \_ \Leftarrow exp : con \mid exp' \leadsto [lbl \triangleright var{=}exp] : [lbl \triangleright var{:}con]} \tag{238}$$

Rule 238: Pattern match against a wildcard.

$$\frac{\Gamma \vdash ty \leadsto con' : \Omega \qquad \Gamma \vdash con \equiv con' : \Omega \qquad \Gamma \vdash pat \Leftarrow exp : con \mid exp' \leadsto mod : sig}{\Gamma \vdash pat : ty \Leftarrow exp : con \mid exp' \leadsto mod : sig} \tag{239}$$

Rule 239: Pattern match against an explicitly-typed pattern.

$$\frac{\Gamma \vdash_{\mathsf{ctx}} \overline{longid} \leadsto path : [\mathrm{mk} \triangleright con' {\to} con'', \mathrm{km} \triangleright con'' {\rightharpoonup} con'] \qquad \Gamma \vdash con \equiv con'' : \Omega \qquad \Gamma \vdash exp' : con' \qquad \Gamma \vdash pat \Leftarrow \mathsf{catch}\ (path.\mathrm{km}\ exp)\ \mathsf{with\ raise}^{con'}\ exp' : con' \mid exp' \leadsto mod : sig}{\Gamma \vdash longid\ pat \Leftarrow exp : con \mid exp' \leadsto mod : sig} \tag{240}$$

Rule 240: Pattern match against a monomorphic (datatype or exception) constructor which carries a value.

$$\frac{\begin{array}{c}\Gamma \vDash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : [\mathrm{mk}{\triangleright}\mathit{con}_{\mathrm{mk}}, \mathrm{km}{\triangleright}\mathit{con}''{\rightharpoonup}\mathsf{Unit}] \\ \Gamma \vdash \mathit{con} \equiv \mathit{con}'' : \Omega \qquad \Gamma \vdash \mathit{exp}' : \mathsf{Unit} \\ \mathit{lbl} \text{ fresh} \qquad \mathit{var} \notin \mathrm{BV}(\Gamma)\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{longid} \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \\ \qquad \left[\mathit{lbl}{\triangleright}\mathit{var}{=}\mathsf{catch}\, \mathit{path}.\mathrm{km}\, \mathit{exp}\, \mathsf{with\ raise}^{\mathsf{Unit}}\, \mathit{exp}'\right] : \\ \qquad [\mathit{lbl}{\triangleright}\mathit{var}{:}\mathsf{Unit}]\end{array}} \tag{241}$$

Rule 241: Pattern match against a constant, monomorphic (datatype or exception) constructor.

$$\frac{\begin{array}{c}\Gamma \vDash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : [\mathrm{mk}{\triangleright}\mathit{sig}_{\mathrm{mk}}, \mathrm{km}{\triangleright}\mathit{sig}_{\mathrm{km}}] \\ \Gamma \vdash \mathit{sig}_{\mathrm{km}} \equiv (\mathit{var}'{:}\mathit{sig}'_{\mathrm{km}}){\rightharpoonup}\mathit{sig}''_{\mathrm{km}} : \mathsf{Sig} \\ \Gamma \vDash_{\mathrm{inst}} \mathit{sig}'_{\mathrm{km}} \rightsquigarrow \mathit{mod}_{\mathit{poly}} : \mathit{sig}_{\mathit{poly}} \\ \Gamma \vdash \mathit{sig}_{\mathrm{km}} \leq \mathit{sig}_{\mathit{poly}} {\rightarrow} [\mathrm{it}{\triangleright}\mathit{con}{\rightharpoonup}\mathit{con}'] : \mathsf{Sig} \\ \Gamma \vdash \mathit{sig}_{\mathrm{mk}} \leq \mathit{sig}_{\mathit{poly}} {\rightarrow} [\mathrm{it}{\triangleright}\mathit{con}'{\rightharpoonup}\mathit{con}] : \mathsf{Sig} \\ \Gamma \vdash \mathit{exp}' : \mathsf{Any} \\ \Gamma \vdash \mathit{pat} \Leftarrow \mathsf{catch}^{\mathit{con}'}\, (((\mathit{path}.\mathrm{km}\, \mathit{mod}_{\mathit{poly}}).\mathrm{it})\, \mathit{exp}) \\ \qquad \mathsf{with\ raise}^{\mathit{con}'}\, \mathit{exp}' : \mathit{con}' \mid \mathit{exp}' \rightsquigarrow \mathit{mod} : \mathit{sig}\end{array}}{\Gamma \vdash \mathit{longid}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{242}$$

Rule 242: Pattern match against a polymorphic datatype constructor carrying a value.

$$\frac{\begin{array}{c}\Gamma \vDash_{\mathrm{ctx}} \overline{\mathit{longid}} \rightsquigarrow \mathit{path} : [\mathrm{mk}{\triangleright}\mathit{sig}_{\mathrm{mk}}, \mathrm{km}{\triangleright}\mathit{sig}_{\mathrm{km}}] \\ \Gamma \vdash \mathit{sig}_{\mathrm{km}} \equiv (\mathit{var}'{:}\mathit{sig}'_{\mathrm{km}}){\rightharpoonup}\mathit{sig}''_{\mathrm{km}} : \mathsf{Sig} \\ \Gamma \vDash_{\mathrm{inst}} \mathit{sig}'_{\mathrm{km}} \rightsquigarrow \mathit{mod}_{\mathit{poly}} : \mathit{sig}_{\mathit{poly}} \\ \Gamma \vdash \mathit{sig}_{\mathrm{km}} \leq \mathit{sig}_{\mathit{poly}} {\rightarrow} [\mathrm{it}{\triangleright}\mathit{con}{\rightharpoonup}\mathsf{Unit}] : \mathsf{Sig} \\ \Gamma \vdash \mathit{sig}_{\mathrm{mk}} \leq \mathit{sig}_{\mathit{poly}} {\rightarrow} [\mathrm{it}{\triangleright}\mathit{con}] : \mathsf{Sig} \\ \Gamma \vdash \mathit{exp}' : \mathsf{Any}\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{longid} \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \\ \qquad \left[\mathit{lbl}{\triangleright}\mathit{var}{:}\mathsf{catch}\, ((\mathit{path}.\mathrm{km}\, \mathit{mod}_{\mathit{poly}}).\mathrm{it})\, \mathit{exp}\, \mathsf{with\ raise}^{\mathsf{Unit}}\, \mathit{exp}'\right] : \\ \qquad [\mathit{lbl}{\triangleright}\mathit{var}{:}\mathsf{Unit}]\end{array}} \tag{243}$$

Rule 243: Pattern match against a constant polymorphic constructor.

$$
\frac{\begin{array}{c}
\mathit{lbl}\ \text{fresh} \qquad \mathit{var} \notin \mathrm{BV}(\Gamma) \\
\Gamma \vdash \mathit{con} \equiv \{\mathit{lbl}'_1 \triangleright \mathit{con}'_1, \cdots, \mathit{lbl}'_k \triangleright \mathit{con}'_k\} : \Omega \\
\{\overline{\mathit{lab}_1}, \cdots, \overline{\mathit{lab}_n}\} \subseteq \{\mathit{lbl}'_1, \cdots, \mathit{lbl}'_n\} \\
\Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathit{con} \vdash \mathit{pat}_1 \Leftarrow \mathit{var}.\overline{\mathit{lab}_1} : \mathit{con}_1 \mid \mathit{exp}' \rightsquigarrow \\
\quad [\mathit{sbnds}_1] : [\mathit{sdecs}_1] \\
\vdots \\
\Gamma, \mathit{lbl} \triangleright \mathit{var}{:}\mathit{con} \vdash \mathit{pat}_n \Leftarrow \mathit{var}.\overline{\mathit{lab}_n} : \mathit{con}_n \mid \mathit{exp}' \rightsquigarrow \\
\quad [\mathit{sbnds}_n] : [\mathit{sdecs}_n]
\end{array}}{\begin{array}{c}
\Gamma \vdash \{\mathit{lab}_1 = \mathit{pat}_1, \ldots, \mathit{lab}_n = \mathit{pat}_n \langle, \ldots \rangle\} \\
\Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \\
[\mathit{lbl} \triangleright \mathit{var}{=}\mathit{exp}, \mathit{sbnds}_1, \ldots, \mathit{sbnds}_n] : \\
[\mathit{lbl} \triangleright \mathit{var}{:}\mathit{exp}, \mathit{sdecs}_1, \ldots, \mathit{sdecs}_n]
\end{array}} \tag{244}
$$

Rule 244: Pattern match against a record of patterns. Because we disallow repeated variables in patterns, the syntactic concatenation of structure here is well-formed.

$$
\frac{\begin{array}{c}
\mathit{lbl}\ \text{fresh} \qquad \mathit{var} \notin \mathrm{BV}(\Gamma) \\
\Gamma \vdash \mathit{pat}_1 \Leftarrow \mathit{var} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow [\mathit{sbnds}_1] : [\mathit{sdecs}_1] \\
\Gamma \vdash \mathit{pat}_2 \Leftarrow \mathit{var} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow [\mathit{sbnds}_2] : [\mathit{sdecs}_2]
\end{array}}{\begin{array}{c}
\Gamma \vdash \mathit{pat}_1\ \texttt{as}\ \mathit{pat}_2 \Leftarrow \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \\
[\mathit{lbl} \triangleright \mathit{var}{=}\mathit{exp}, \mathit{sbnds}_1, \mathit{sbnds}_2] : \\
[\mathit{lbl} \triangleright \mathit{var}{:}\mathit{con}, \mathit{sdecs}_1, \mathit{sdecs}_2]
\end{array}} \tag{245}
$$

Rule 245: Pattern match against two patterns simultaneously.

$$
\frac{\Gamma \vdash \mathit{pat} \Leftarrow \mathsf{get}\ \mathit{exp} : \mathit{con} \mid \mathit{exp}' \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{ref}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{Ref} \mid \mathit{exp}' \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{246}
$$

Rule 246: Pattern match involving implicit dereferencing of a `ref` cell.

6.9 Coercion Compilation

$$\boxed{\begin{array}{r} decs \vdash_{\text{sub}} sig_0 \preceq sig \leadsto \\ \lambda var_0{:}sig_0.mod : \\ (var_0{:}sig_0) \rightarrow sig' \end{array}}$$

This judgment should be read as "the coercion functor that copies the components appearing in *sig* from a structure matching signature sig_0 is $\lambda(var_0{:}sig_0).mod$ and has signature $(var_0{:}sig_0) \rightarrow sig'$; this signature maximizes type propagation."

$$\frac{}{decs \vdash_{\text{sub}} sig_0 \preceq [] \leadsto \lambda var_0{:}sig_0.\,[] : (var_0{:}sig_0) \rightarrow []} \tag{247}$$

Rule 247: Coercion to forget all components.

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \leadsto lbls : con' \\ decs \vdash con \equiv con' : \Omega \\ decs, var{:}con \vdash_{\text{sub}} sig_0 \preceq [sdecs] \leadsto \\ \quad \lambda var_0{:}sig_0.\,[sbnds] : (var_0{:}sig_0) \rightarrow [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var{:}con, sdecs] \leadsto \\ \quad \lambda var_0{:}sig_0.\,[lbl \triangleright var{=}var_0.lbls, sbnds] : \\ \quad (var_0{:}sig_0) \rightarrow [lbl \triangleright var{:}con, sdecs'] \end{array}} \tag{248}$$

Rule 248: Coercion of a monomorphic value specification to a monomorphic value specification.

$$\frac{\begin{array}{c}\left\{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\rm sig} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0{:}sig_0 \vdash_{\rm sig} lbl \rightsquigarrow lbls : [\text{mk}\triangleright sig, \text{km}\triangleright sig_{\rm km}] \end{array}\right\} \\ decs, var_1{:}sig_1 \vdash_{\rm inst} sig_{poly} \rightsquigarrow mod_{poly} : sig'_{poly} \\ decs, var_1{:}sig_1 \vdash sig \leq sig'_{poly} \rightarrow [\text{it}\triangleright con] : \mathsf{Sig} \\ decs, var{:}(var_1{:}sig_1) \rightarrow [\text{it}\triangleright con] \vdash_{\rm sub} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\, [sbnds] : (var_0{:}sig_0) \rightarrow [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\rm sub} sig_0 \preceq [lbl \triangleright var{:}(var_1{:}sig_1) \rightarrow [\text{it}\triangleright con]\,, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0. \left[lbl \triangleright var{=}\lambda var_1{:}sig_1. \left\{\begin{array}{c} var_0.lbls \\ \text{or} \\ var_0.lbls.\text{mk} \end{array}\right\} var_{poly}, sbnds \right] : \\ \quad (var_0{:}sig_0) \rightarrow [lbl \triangleright var{:}(var_1{:}sig_1) \rightarrow [\text{it}\triangleright con]\,, sdecs'] \end{array}} \tag{249}$$

Rule 249: Coercion of a polymorphic value specification to a polymorphic value specification; this may involve implicit polymorphic instantiation. (The rule also handles matching polymorphic datatype constructors to `val` specifications.)

$$\frac{\begin{array}{c}\left\{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\rm sig} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0{:}sig_0 \vdash_{\rm sig} lbl \rightsquigarrow lbls : [\text{mk}\triangleright sig, \text{km} \sim sig_{\rm km}] \end{array}\right\} \\ decs \vdash_{\rm inst} sig_{poly} \rightsquigarrow mod_{poly} : sig'_{poly} \\ decs \vdash sig \leq sig'_{poly} \rightarrow [\text{it}\triangleright con] : \mathsf{Sig} \\ decs, var{:}con \vdash_{\rm sub} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\, [sbnds] : (var_0{:}sig_0) \rightarrow [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\rm sub} sig_0 \preceq [lbl \triangleright var{:}con, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0. \left[lbl \triangleright var{=}\lambda var_1{:}sig_1.(\left\{\begin{array}{c} var_0.lbls \\ \text{or} \\ var_0.lbls.\text{mk} \end{array}\right\} var_{poly}).\text{it}, sbnds \right] : \\ \quad (var_0{:}sig_0) \rightarrow [lbl \triangleright var{:}con, sdecs'] \end{array}} \tag{250}$$

Rule 250: Coercion of a polymorphic value (or datatype constructor) to match a monomorphic value specification.

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\rm sig} lbl \rightsquigarrow lbls : knd \\ \langle decs, var_0{:}sig_0 \vdash var_0.lbls \equiv con : knd \rangle \\ decs, var{:}knd{=}var_0.lbls \vdash_{\rm sub} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\, [sbnds] : (var_0{:}sig_0) \rightarrow [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\rm sub} sig_0 \preceq [lbl \triangleright var{:}knd\langle{=}con\rangle, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0.\, [lbl \triangleright var{=}var_0.lbls, sbnds] : \\ \quad (var_0{:}sig_0) \rightarrow [lbl \triangleright var{:}knd{=}var_0.lbls, sdecs'] \end{array}} \tag{251}$$

Rule 251: Coercion of a type component to a `type` specification.

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\mathrm{sig}} lbl \rightsquigarrow lbls : \Omega \\ \langle decs, var_0{:}sig_0 \vdash var_0.lbls \equiv con : \Omega \rangle \\ decs, var_0{:}sig_0 \vdash_{\mathrm{eq}} var_0.lbls \rightsquigarrow exp_{\mathrm{v}} \\ decs, var{:}knd{=}var_0.lbls \vdash_{\mathrm{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\,[sbnds] : (var_0{:}sig_0) {\to} [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{sub}} sig_0 \preceq [lbl' {\triangleright} [lbl {\triangleright} var{:}\Omega\langle{=}con\rangle, \mathrm{eq}{\triangleright} var \times var {\rightharpoonup} \mathsf{Bool}]\,, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0.\,[lbl' {\triangleright} [lbl {\triangleright} var{=}var_0.lbls, \mathrm{eq}{\triangleright} exp_{\mathrm{v}}]\,, sbnds] : \\ \quad (var_0{:}sig_0) {\to} [lbl' {\triangleright} [lbl {\triangleright} var{:}\Omega{=}var_0.lbls, \mathrm{eq}{\triangleright} var \times var {\rightharpoonup} \mathsf{Bool}]\,, sdecs'] \end{array}} \tag{252}$$

Coercion of a type declaration corresponding to an `eqtype` signature specification involving no type variables. We invoke the equality compiler to create an equality function for the type being copied.

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\mathrm{sig}} lbl \rightsquigarrow lbls : \Omega^n {\Rightarrow} \Omega \\ \langle decs, var_0{:}sig_0 \vdash var_0.lbls \equiv con : \Omega^n {\Rightarrow} \Omega \rangle \\ con' = var(var'.lbl'_1.lbl_1, \cdots, var'.lbl'_n.lbl_n) \\ decs, var_0{:}sig_0, var'{:}sig' \vdash_{\mathrm{eq}} con' \rightsquigarrow exp_{\mathrm{v}} \\ decs, var{:}knd{=}var_0.lbls \vdash_{\mathrm{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\,[sbnds] : (var_0{:}sig_0) {\to} [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{sub}} sig_0 \preceq [lbl' {\triangleright} [lbl {\triangleright} var{:}\Omega^n {\Rightarrow} \Omega\langle{=}con\rangle, \mathrm{eq}{\triangleright} var'{:}sig' {\to} [\mathrm{it}{\triangleright} con' \times con' {\rightharpoonup} \mathsf{Bool}]]\,, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0.\,[lbl' {\triangleright} [lbl {\triangleright} var{=}var_0.lbls, \mathrm{eq}{\triangleright} \lambda var'{:}sig'.\,[\mathrm{it}{\triangleright} exp_{\mathrm{v}}]]\,, sbnds] : \\ \quad (var_0{:}sig_0) {\to} [lbl' {\triangleright} [lbl {\triangleright} var{:}\Omega^n {\Rightarrow} \Omega\langle{=}var_0.lbls\rangle, \mathrm{eq}{\triangleright} var'{:}sig' {\to} [\mathrm{it}{\triangleright} con' \times con' {\rightharpoonup} \mathsf{Bool}]]\,, sdecs'] \end{array}} \tag{253}$$

Coercion of a type declaration correspond to a `eqtype` signature specification involving type variables (i.e., `eqtype ('a,'b) foo`).

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\mathrm{sig}} lbl \rightsquigarrow lbls : sig_1 \\ decs \vdash_{\mathrm{sub}} sig_1 \preceq sig \rightsquigarrow \lambda var_1{:}sig_1.mod : (var_1{:}sig_1) {\to} sig'' \\ decs, var{:}sig \vdash_{\mathrm{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ \lambda var_0{:}sig_0.\,[sbnds] : (var_0{:}sig_0) {\to} [sdecs'] \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{sub}} sig_0 \preceq [lbl {\triangleright} var{:}sig, sdecs] \rightsquigarrow \\ \quad \lambda var_0{:}sig_0.\,[lbl {\triangleright} var{=}\lambda var_1{:}sig_1.mod\; var_0.lbls, sbnds] : \\ \quad (var_0{:}sig_0) {\to} [lbl {\triangleright} var{:}sig'', sdecs'] \end{array}} \tag{254}$$

Rule 254: Coercion of a module component.

$$\frac{\begin{array}{c} decs \vdash_{\mathrm{sub}} sig_2 \preceq sig_1 \rightsquigarrow mod_3 : sig_3 \\ decs \vdash_{\mathrm{sub}} sig'_1 \preceq sig'_2 \rightsquigarrow mod_4 : sig_4 \\ decs, var_0{:}var_1{:}sig_1 {\rightharpoonup} sig'_1, var_2{:}sig_2 \vdash mod_4\,(var_0\,(mod_3\; var_2)) : sig \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{sub}} var_1{:}sig_1 {\rightharpoonup} sig'_1 \preceq var_2{:}sig_2 {\rightharpoonup} sig'_2 \rightsquigarrow \\ \quad \lambda var_0{:}var_1{:}sig_1 {\rightharpoonup} sig'_1.\lambda var_2{:}sig_2.mod_4\,(var_0\,(mod_3\; var_2)) : \\ \quad (var_0{:}var_1{:}sig_1 {\rightharpoonup} sig'_1) {\to} var_2{:}sig_2 {\rightharpoonup} sig \end{array}} \tag{255}$$

$$\frac{\begin{array}{c} decs \vdash_{\mathsf{sub}} sig_2 \preceq sig_1 \leadsto mod_3 : sig_3 \\ decs \vdash_{\mathsf{sub}} sig_1' \preceq sig_2' \leadsto mod_4 : sig_4 \\ decs, var_0{:}var_1{:}sig_1{\to}sig_1', var_2{:}sig_2 \vdash mod_4\,(var_0\,(mod_3\; var_2)) : sig \end{array}}{\begin{array}{l} decs \vdash_{\mathsf{sub}} var_1{:}sig_1{\to}sig_1' \preceq var_2{:}sig_2{\to}sig_2' \leadsto \\ \qquad \lambda var_0{:}var_1{:}sig_1{\to}sig_1'.\lambda var_2{:}sig_2.mod_4\,(var_0\,(mod_3\; var_2)) : \\ \qquad (var_0{:}var_1{:}sig_1{\to}sig_1'){\to}var_2{:}sig_2{\to}sig \end{array}} \tag{256}$$

Rules 255, 256: Coercions for functor subtyping. By the IL subtyping rules, we can use Rule 255 to coerce a total functor to a partial functor.

6.10 Lookup Rules

The lookup rules specify the order in which translation contexts and IL signatures are searched.

- To prevent an explosion of rules, the metavariable *class* is used to denote a type, signature, or kind as appropriate.
- Any IL structure label starred with an asterisk (e.g., lbl^*) is treated specially by the lookup, which checks inside the structure. That is, when looking for an identifier in a context, we also look *inside* starred structure declarations. If the identifier is found inside such a structure, the full path to the identifier through the open structure is returned.
- The lookup of signatures in a context is treated specially by returning the signature rather than the path to the signature.
- Any type or signature returned by a lookup will be valid with respect to the ambient context.

6.10.1 Context Lookup

$$\boxed{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path/sig : class}$$

Main rules for looking up identifiers in a translation context.

$$\frac{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : con}{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path : con} \tag{257}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : knd}{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path : knd} \tag{258}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : sig}{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path : sig} \tag{259}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow sig \qquad \Gamma \vdash sig : \mathsf{Sig}}{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow sig : \mathsf{Sig}} \tag{260}$$

$$\boxed{\Gamma \vdash_{\mathsf{ctx}} lbls \rightsquigarrow path/sig}$$

"Utility" rules used to look up identifiers in a translation context.

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}con \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow var'} \tag{261}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}con \vdash_{\mathsf{ctx}} lbl \rightsquigarrow path} \tag{262}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}knd\langle = con\rangle \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow var} \tag{263}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}knd\langle = con\rangle \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path} \tag{264}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow var} \tag{265}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path} \tag{266}$$

$$\frac{sig \vdash_{\mathsf{sig}} lbl' \rightsquigarrow path}{\Gamma, lbl^{*} \triangleright var{:}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow var.path} \tag{267}$$

$$\frac{sig \vdash_{\mathsf{sig}} lbl' \not\rightsquigarrow \qquad \Gamma \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl^{*} \triangleright var{:}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path} \tag{268}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}\mathsf{Sig}{=}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow sig} \tag{269}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}\mathsf{Sig}{=}sig \vdash_{\mathsf{ctx}} lbl' \rightsquigarrow path} \tag{270}$$

$$\frac{\Gamma \vdash_{\mathsf{ctx}} lbl \rightsquigarrow path : sig \qquad \Gamma, path{:}sig \vdash_{\mathsf{sig}} lbls \rightsquigarrow lbls' : class}{\Gamma \vdash_{\mathsf{ctx}} lbl.lbls \rightsquigarrow path.lbls'} \tag{271}$$

6.10.2 Signature Lookup

$$\boxed{decs, path{:}sig \vdash_{\mathsf{sig}} lbls \rightsquigarrow lbls' : class}$$

Main rules for looking up identifiers in a signature.

$$\frac{decs \vdash path : sig \qquad sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' \qquad decs \vdash path.lbls' : con}{decs, path{:}sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' : con} \tag{272}$$

$$\frac{decs \vdash path : sig \qquad sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' \qquad decs \vdash path.lbls' : sig'}{decs, path{:}sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' : sig'} \tag{273}$$

$$\frac{decs \vdash path : sig \qquad sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' \qquad decs \vdash path.lbls' : knd}{decs, path{:}sig \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' : knd} \tag{274}$$

$$\frac{\begin{array}{c} decs \vdash path : sig \\ decs, path{:}lbls \vdash_{\mathsf{sig}} lbl \rightsquigarrow lbls' : sig' \\ decs, path.lbls'{:}sig' \vdash_{\mathsf{sig}} lbls \rightsquigarrow lbls'' : class \end{array}}{decs, path{:}sig \vdash_{\mathsf{sig}} lbl.lbls \rightsquigarrow lbls'.lbls'' : class} \tag{275}$$

$$\boxed{sig \vdash_{\mathrm{sig}} lbl \rightsquigarrow lbls'}$$

"Utility" rules used to look up identifiers in a signature.

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}con] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbl} \tag{276}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}con] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls} \tag{277}$$

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}knd\langle = con\rangle] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbl} \tag{278}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}knd\langle = con\rangle] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls} \tag{279}$$

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}sig] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbl} \tag{280}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}sig] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls} \tag{281}$$

$$\frac{sig \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var{:}sig] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbl^*.lbls} \tag{282}$$

$$\frac{sig \vdash_{\mathrm{sig}} lbl' \not\rightsquigarrow \qquad [sdecs] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var{:}sig] \vdash_{\mathrm{sig}} lbl' \rightsquigarrow lbls} \tag{283}$$

6.11 Signature Patching

where type

$$\boxed{sig \vdash_{\mathrm{wt}} lbls := con : knd \rightsquigarrow sig' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature *sig* the fact that the (abstract) type component selected by *lbls* is equal to *con* : *knd*, we get the signature *sig'*."

$$\frac{\begin{array}{c}\mathrm{FV}(con) \cap \mathrm{BV}(sdecs) = \emptyset \\ sig = [sdecs, lbl{\triangleright}var{:}knd, sdecs']\end{array}}{sig \vdash_{\mathrm{wt}} lbl := con : knd \rightsquigarrow [sdecs, lbl{\triangleright}var{:}knd{=}con, sdecs'] : \mathsf{Sig}} \tag{284}$$

$$\frac{\begin{array}{c}\mathrm{FV}(con) \cap \mathrm{BV}(sdecs) = \emptyset \\ sig = [sdecs, lbl{\triangleright}var{:}sig', sdecs'] \\ sig' \vdash_{\mathrm{wt}} lbls := con : knd \rightsquigarrow sig'' : \mathsf{Sig}\end{array}}{sig \vdash_{\mathrm{wt}} lbl.lbls := con : knd \rightsquigarrow [sdecs, lbl{\triangleright}var{:}sig'', sdecs'] : \mathsf{Sig}} \tag{285}$$

sharing

$$\boxed{sig \vdash_{\mathrm{sh}} lbls_1 := lbls_2 : knd \rightsquigarrow sig' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature *sig* the fact that the (abstract) type components of kind *knd* selected by $lbls_1$ and $lbls_2$ are equal, we get the signature *sig'*."

$$\frac{sig = [sdecs, lbl'{\triangleright}var'{:}knd, sdecs', lbl{\triangleright}var{:}knd, sdecs'']}{\begin{array}{l}sig \vdash_{\mathrm{sh}} lbl := lbl' : knd \rightsquigarrow \\ \qquad [sdecs, lbl'{\triangleright}var'{:}knd, sdecs', lbl{\triangleright}var{:}knd{=}var', sdecs'] : \mathsf{Sig}\end{array}} \tag{286}$$

$$\frac{\begin{array}{c}sig = [sdecs, lbl'{\triangleright}var'{:}knd, sdecs', lbl{\triangleright}var{:}sig, sdecs''] \\ sig \vdash_{\mathrm{wt}} lbls := var' : knd \rightsquigarrow sig' : \mathsf{Sig}\end{array}}{\begin{array}{l}sig \vdash_{\mathrm{sh}} lbl.lbls := lbl' : knd \rightsquigarrow \\ \qquad [sdecs, lbl'{\triangleright}var'{:}knd, sdecs', lbl{\triangleright}var{:}sig', sdecs'] : \mathsf{Sig}\end{array}} \tag{287}$$

$$\frac{\begin{array}{c}sig = [sdecs, lbl'{\triangleright}var'{:}sig', sdecs', lbl{\triangleright}var{:}sig'', sdecs''] \\ sig'' \vdash_{\mathrm{wt}} lbls' := var'.lbls' : knd \rightsquigarrow sig''' : \mathsf{Sig}\end{array}}{\begin{array}{l}sig \vdash_{\mathrm{sh}} lbl.lbls := lbl'.lbls' : knd \rightsquigarrow \\ \qquad [sdecs, lbl'{\triangleright}var'{:}sig', sdecs', lbl{\triangleright}var{:}sig''', sdecs''] : \mathsf{Sig}\end{array}} \tag{288}$$

$$\frac{\begin{array}{c}sig = [sdecs, lbl{\triangleright}var{:}sig', sdecs'] \\ sig' \vdash_{\mathrm{sh}} lbls := lbls' : knd \rightsquigarrow sig'' : \mathsf{Sig}\end{array}}{sig \vdash_{\mathrm{sh}} lbl.lbls := lbl.lbls' : knd \rightsquigarrow [sdecs, lbl{\triangleright}var{:}sig'', sdecs'] : \mathsf{Sig}} \tag{289}$$

6.12 Overloading

For each overloaded identifier we wish to add to the EL, we add one translation rule for each overloaded type. For example, we may want to overload the EL identifier `+` for addition at types `int` and `real`. In this case, we would add two rules along the lines of:

$$\frac{}{\Gamma \vdash \mathsf{+} \leadsto +_{\mathsf{Int}} : \mathsf{Int} \times \mathsf{Int} \rightharpoonup \mathsf{Int}}$$

and

$$\frac{}{\Gamma \vdash \mathsf{+} \leadsto +_{\mathsf{Float}} : \mathsf{Float} \times \mathsf{Float} \rightharpoonup \mathsf{Float}},$$

where $+_{\mathsf{Int}}$ and $+_{\mathsf{Float}}$ are the two IL primitive addition functions.

7 Theorems[1]

7.1 Properties of the IL

In preparation for the proof of soundness of the IL with respect to the dynamic semantics we will need to state some technical lemmas.

We define two states to be *equivalent*, written

$$(\Delta, \sigma, E, \mathit{phrase}) \cong (\Delta', \sigma', E', \mathit{phrase}'),$$

if the two states are equal componentwise up to consistent renaming of the locations and exception tags appearing in Δ. This is clearly an equivalence relation on states.

Lemma 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 lemma states that IL judgments are preserved under substitution of values for free variables in a typing judgement, where a value is defined syntactically in Figure 6 to be a phrase in evaluated form.

Proposition 2 (Substitution)

1. *If $\mathit{decs}, \mathit{var}{:}\mathit{class}, \mathit{decs}' \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \mathit{knd}$ and $\mathit{decs} \vdash \mathit{val} : \mathit{class}$ then*
   $\mathit{decs}, \{\mathit{val}/\mathit{var}\}\mathit{decs}' \vdash$.
   $\{\mathit{val}/\mathit{var}\}\mathit{con}_1 \equiv \{\mathit{val}/\mathit{var}\}\mathit{con}_2 : \mathit{knd}$
2. *If $\mathit{decs}, \mathit{var}{:}\mathit{class}, \mathit{decs}' \vdash \mathit{sig}_1 \equiv \mathit{sig}_2 : \mathsf{Sig}$ and $\mathit{decs} \vdash \mathit{val} : \mathit{class}$ then*
   $\mathit{decs}, \{\mathit{val}/\mathit{var}\}\mathit{decs}' \vdash$.
   $\{\mathit{val}/\mathit{var}\}\mathit{sig}_1 \equiv \{\mathit{val}/\mathit{var}\}\mathit{sig}_2 : \mathsf{Sig}$
3. *If $\mathit{decs}, \mathit{var}{:}\mathit{class}, \mathit{decs}' \vdash \mathit{sig}_1 \leq \mathit{sig}_2 : \mathsf{Sig}$ and $\mathit{decs} \vdash \mathit{val} : \mathit{class}$ then*
   $\mathit{decs}, \{\mathit{val}/\mathit{var}\}\mathit{decs}' \vdash$.
   $\{\mathit{val}/\mathit{var}\}\mathit{sig}_1 \leq \{\mathit{val}/\mathit{var}\}\mathit{sig}_2 : \mathsf{Sig}$
4. *If $\mathit{decs}, \mathit{var}{:}\mathit{class}, \mathit{decs}' \vdash \mathit{exp} : \mathit{con}$ and $\mathit{decs} \vdash \mathit{val} : \mathit{class}$ then*
   $\mathit{decs}, \{\mathit{val}/\mathit{var}\}\mathit{decs}' \vdash \{\mathit{val}/\mathit{var}\}\mathit{exp} : \{\mathit{val}/\mathit{var}\}\mathit{con}.$
5. *If $\mathit{decs}, \mathit{var}{:}\mathit{class}, \mathit{decs}' \vdash \mathit{mod} : \mathit{sig}$ and $\mathit{decs} \vdash \mathit{val} : \mathit{class}$ then*
   $\mathit{decs}, \{\mathit{val}/\mathit{var}\}\mathit{decs}' \vdash \{\mathit{val}/\mathit{var}\}\mathit{mod} : \{\mathit{val}/\mathit{var}\}\mathit{sig}.$

[1]The proofs presented here are only sketches; the SML '96 language is far too large to check every case by hand in a reasonable amount of time. Future work may involve attempting to check these proofs with an automated system.

6. *If* $decs, var{:}class, decs' \vdash con : knd$ and $decs \vdash val : sig'$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}con : knd$.

7. *If* $decs, var{:}class, decs' \vdash exp \downarrow con$ and $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}exp \downarrow \{val/var\}con$.

8. *If* $decs, var{:}class, decs' \vdash mod \downarrow sig$ and $decs \vdash val : class$ *then* $decs, \{val/var\}decs' \vdash \{val/var\}mod \downarrow \{val/var\}sig$.

Proof (sketch): [By induction on the derivation of the first premises; due to space constraints we show only two sample cases]

- Case: The derivation ends with Typing Rule 50:

$$\frac{\begin{array}{c} decs, var{:}class, decs' \vdash exp : con_2 \rightharpoonup con \\ decs, var{:}class, decs' \vdash exp' : con_2 \end{array}}{decs, var{:}class, decs' \vdash exp\ exp' : con}$$

  and $decs \vdash val : class$. By the inductive hypotheses,

$$\begin{array}{l} decs, \{val/var\}decs' \vdash \{val/var\}exp : \\ \qquad (\{val/var\}con_2) \rightharpoonup (\{val/var\}con) \end{array}$$

  and

$$decs, \{val/var\}decs' \vdash \{val/var\}exp' : \{val/var\}con_2.$$

  Therefore from Typing Rule 50, we can conclude

$$decs, \{val/var\}decs' \vdash \{exp_{\mathrm{v}}/var\}(exp\ exp') : con.$$

- Case: the derivation ends with Typing Rule 29:

$$\frac{\begin{array}{c} decs, var{:}class, decs' \vdash mod_{\mathrm{v}} : [lbl_1 \triangleright dec_1, \cdots, lbl_m \triangleright dec_m, \\ \qquad\qquad lbl \triangleright var{:}knd{=}con_2, sdecs] \\ decs, var{:}class, decs', dec_1, \cdots, dec_m \vdash con \equiv con_2 : knd \\ decs, var{:}class, decs' \vdash con : knd \end{array}}{decs, var{:}class, decs' \vdash mod_{\mathrm{v}}.lbl \equiv con : knd}$$

  and $decs \vdash val : class$. By the using properties 5, 1, and 6 of the induction hypothesis, and Typing Rule 29, we have

$$\begin{array}{l} decs, \{val/var\}decs' \vdash \{val/var\}mod_{\mathrm{v}}.lbl \equiv \\ \qquad\qquad \{val/var\}con : knd. \end{array}$$

■

A context C is a phrase with a single *hole*, written []. We write $C[phrase]$ for the result of filling the hole in C with *phrase*, possibly incurring variable capture. A stack of frames E determines a context $\widehat{E}$ in the obvious way: $\widehat{[]}$ is the context [], and $\widehat{E \circ F}$ is the context $\widehat{E}[F]$.

Proposition 3 (Decomposition & Replacement)

1. *If* $decs \vdash \widehat{E}[exp] : con$ *and* exp *is closed, then* $decs \vdash exp : con'$ *for some type* con'. *Furthermore, if* $decs \vdash exp' : con'$ *where* exp' *is closed, then* $decs \vdash \widehat{E}[exp'] : con$.

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

3. *If* $decs \vdash \widehat{E}[con'] : con$ *and* con' *is closed, then* $decs \vdash con : knd$ *for some kind* knd. *Furthermore, if* $decs \vdash con'' : knd$ *where* con'' *is closed, then* $decs \vdash \widehat{E}[con''] : con$.

Proof (sketch): [By induction on E]

If $E = []$ then the proposition is trivial, since $\widehat{E}[phrase] = phrase$. Hence, assume $E = E' \circ F$. Due to space constraints, we show just one prototypical case.

- Case: $F = []\, exp_2$. Then $\widehat{E}[exp] = \widehat{E'}[exp\; exp_2]$. By the inductive hypothesis, we have $decs \vdash exp\; exp_2 : con'$ for some con'. Inspection of the typing rules shows that this derivation must contain an application of Rule 50, so that by inversion $decs \vdash exp : con_2 \rightharpoonup con'$ and $decs \vdash exp_2 : con_2$ for some type con_2. Furthermore, if $decs \vdash exp' : con_2 \rightharpoonup con'$ then $decs \vdash exp'\; exp : con'$ and by the inductive hypothesis we have $decs \vdash \widehat{E'}[exp'\; exp_2] : con$; that is, $decs \vdash \widehat{E}[exp'] : con$.

■

Following Harper [Har93], we say that a store σ is well-formed with respect to a context Δ, written $\Delta \vdash \sigma$, if

$$\forall x \in \mathrm{BV}(\Delta) : \Delta \vdash \sigma(x) : \Delta(x)$$

This formulation of store typing avoids the need for complex maximal fixed point constructions [Tof90]. (An essentially similar observation was made by Wright and Felleisen [WF91].)

Fix a base type *ans* of *answers* to which a complete, closed program might evaluate.[1] We can say that a machine state is well-formed, written

$$\vdash (\Delta, \sigma, E, phrase),$$

if and only if $\Delta \vdash \widehat{E}[phrase] : ans$, $phrase$ is closed, and $\Delta \vdash \sigma$.

The first component of soundness is that the preservation of well-formedness by the dynamic semantics.

Proposition 4 (Preservation)
If $\vdash (\Delta, \sigma, E, phrase)$ *and*

$$(\Delta, \sigma, E, phrase) \hookrightarrow (\Delta', \sigma', E', phrase')$$

then $\vdash (\Delta', \sigma', E', phrase')$.

[1] 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.

Proof (sketch):
If the transition is a search rule then preservation follows trivially, for $\Delta = \Delta'$, $\sigma = \sigma'$, and $\widehat{E}[phrase] = \widehat{E'}[phrase']$. Hence we need only consider the reduction rules. We show only one sample case here.

- Case: Transition Rule 150:

$$(\Delta, \sigma, E \circ (\pi_{lbl}\,[\,]), \{rbnds_{\mathrm{v}}, lbl{\triangleright}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}) \hookrightarrow \\ (\Delta, \sigma, E, exp_{\mathrm{v}})$$

  Then $\widehat{E \circ (\pi_{lbl}\,[\,])}[\{rbnds_{\mathrm{v}}, lbl{\triangleright}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}] = \widehat{E}[\pi_{lbl}\,\{rbnds_{\mathrm{v}}, lbl{\triangleright}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\}]$. By the well-formedness assumption and Decomposition,
  $decs \vdash \pi_{lbl}\,\{rbnds_{\mathrm{v}}, lbl{\triangleright}exp_{\mathrm{v}}, rbnds_{\mathrm{v}}'\} : con'$ for some type con'. Since this judgement must be proven using Typing Rules 54 and 53, by inversion we have that
  $decs \vdash exp_{\mathrm{v}} : con'$. Since exp_{v} is must by closed by the well-formedness assumption and $decs \vdash \widehat{E}[exp_{\mathrm{v}}] : ans$ by Replacement, well-formedness is preserved.

■

The second component of soundness is that evaluation can never "get stuck": if a well-formed state is not terminal, 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.

Lemma 5 (Canonical Forms)

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

If con' is of the form...	*then exp_{v}' is of the form...*
$con_1 \rightharpoonup con_2$	$\mathsf{fix}\, fbnds\, \mathsf{in}\, var'\, \mathsf{end}$
$con_1 \rightarrow con_2$	$\mathsf{fix}\, fbnds\, \mathsf{in}\, var'\, \mathsf{end}$
$\{lbl_1{\triangleright}con_1, \cdots, lbl_n{\triangleright}con_n\}$	$\{lbl_1{\triangleright}exp_{\mathrm{v}1}, \cdots, lbl_n{\triangleright}exp_{\mathrm{v}n}\}$
$\Sigma\,(con_1, \cdots, con_n)$	$\mathsf{inj}_i^{\Sigma(con_1,\cdots,con_n)}\; exp_{\mathrm{v}}$
$\mu_i\; con$	$\mathsf{roll}^{\mu_i\; con}\; exp_{\mathrm{v}}$
Any	$\mathsf{tag}_{tag}\; exp_{\mathrm{v}}$
$con\; \mathsf{Ref}$	loc
base type	$scon$

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

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

Proof: By inspection of the applicable typing rules for values. For example, consider the case in which $\Delta \vdash exp_{\mathrm{v}}' : con'$ where con' is a record type. There are numerous rules that allow the conclusion that an expression has a record type (application, record projection, etc.) but only one—the rule for record expressions—applies in the case of a closed value. This is the only possible form for exp_{v}. The other cases are analogous. ■

Proposition 6 (Progress)
If $\vdash \Sigma$ then either Σ is terminal or there exists a state Σ' such that $\Sigma \hookrightarrow \Sigma'$.

Proof (sketch):
There are three cases:

1. Σ is terminal.

2. $\Sigma = (\Delta, \sigma, \mu, E, phrase)$ where *phrase* is not a value. By inspection, for every possible phrase syntax there is an applicable rule in the dynamic semantics (a reduction rule for $\mathsf{new\_tag}[con]$ and search rules otherwise).

3. $\Sigma = (\Delta, \sigma, \mu, E \circ F, val)$. This has one subcase for each possible frame F. We show two such cases:

   - Case: $F = (exp_{\mathrm{v}}\,[\,])$. By well-formedness and Decomposition, $\Delta \vdash exp_{\mathrm{v}}\; val : con'$ for some type con'. By inversion of Rule 50, we have that $\Delta \vdash exp_{\mathrm{v}} : con_2 \rightharpoonup con'$ and $\Delta \vdash val : con_2$, so that *val* is an expression value. By Canonical Forms, exp_{v} must be a fix form, so that Transition Rule 149 applies.

   - Case: $F = (\mathsf{get}\,[\,])$. By well-formedness and Decomposition, $\Delta \vdash \mathsf{get}\; val : con'$ for some type con'. By inversion of Rule 59 we have that $\Delta \vdash val : con'\,\mathsf{Ref}$. By Canonical Forms, *val* is a location, and by well-formedness $val \in \mathrm{BV}(\sigma)$. Therefore, Transition Rule 153 applies.

■

7.2 Properties of the Elaborator

The minimal requirement for the elaborator is that the elaboration of EL code yields well-formed IL code:

Proposition 7 (Well-formedness)
The following propositions hold:

1. *If* $\Gamma \vdash expr \leadsto exp : con$ *then* $\Gamma \vdash exp : con$.

2. *If* $\Gamma \vdash match \leadsto exp : con$ *then* $\Gamma \vdash exp : con$.

3. *If* $\Gamma \vdash strdec \leadsto mod : sig$ *then* $\Gamma \vdash mod : sig$.

4. *If* $\Gamma \vdash strexp \leadsto mod : sig$ *then* $\Gamma \vdash mod : sig$.

5. *If* $\Gamma \vdash spec \leadsto [sdecs] : \mathsf{Sig}$ *then* $\Gamma \vdash [sdecs] : \mathsf{Sig}$.

6. *If* $\Gamma \vdash sigexp \leadsto sig : \mathsf{Sig}$ *then* $\Gamma \vdash sig : \mathsf{Sig}$.

7. *If* $\Gamma \vdash ty \leadsto con : \Omega$ *then* $\Gamma \vdash con : \Omega$.

8. *If* $\Gamma \vdash tybind \leadsto mod : sig$ *then* $\Gamma \vdash mod : sig$.

9. *If* $\Gamma \vdash datbind \leadsto mod : sig$ *then* $\Gamma \vdash mod : sig$.

10. *If* $\Gamma \vdash_{\mathrm{ctx}} lbls \leadsto path : class$ *then* $\Gamma \vdash path : class$.

11. *If* $\Gamma \vdash_{\mathrm{ctx}} lbls \leadsto sig : \mathsf{Sig}$ *then* $\Gamma \vdash sig : \mathsf{Sig}$.

12. *If* $\Gamma, path{:}sig \vdash_{\mathrm{sig}} lbls \leadsto lbls : class$ *then* $\Gamma \vdash path.lbls : class$.

13. *If* $decs \vdash_{\mathrm{inst}} [sdecs] \leadsto [sbnds_{\mathrm{v}}] : [sdecs']$ *then* $decs \vdash sbnds_{\mathrm{v}} : [sbnds_{\mathrm{v}}]\,[sdecs']$ *and* $decs \vdash [sdecs'] \leq [sdecs] : \mathsf{Sig}$.

14. *If* $\Gamma \vdash pat \Leftarrow exp : con \mid exp' \leadsto mod : sig$ *then* $\Gamma \vdash mod : sig$.

15. *If* $decs \vdash_{\mathrm{eq}} con \leadsto exp_{\mathrm{v}}$ *then* $decs \vdash exp_{\mathrm{v}} : con \times con \rightharpoonup \mathsf{Bool}$.

16. *If* $decs \vdash_{\mathrm{sub}} sig_0 \preceq sig \leadsto \lambda var_0{:}sig_0.mod : (var_0{:}sig_0) \rightarrow sig'$ *then* $\Gamma \vdash (\lambda var_0{:}sig_0.mod) : (var_0{:}sig_0) \rightarrow sig'$.

17. *If* $sig \vdash_{\mathrm{wt}} lbls := con : knd \leadsto sig' : \mathsf{Sig}$ *and* $\Gamma \vdash sig : \mathsf{Sig}$ *then* $\Gamma \vdash sig' : \mathsf{Sig}$.

18. *If* $sig \vdash_{\mathrm{sh}} lbls := lbls' : knd \leadsto sig' : \mathsf{Sig}$ *and* $\Gamma \vdash sig : \mathsf{Sig}$ *then* $\Gamma \vdash sig' : \mathsf{Sig}$.

Proof: This follows by induction on the derivation of the first premises. ■

To claim that we have *defined* a language, we need elaboration to be a "function" in the sense that all possible elaborations of an EL program yield "equivalent" IL programs.

Proposition 8 (Coherence)
If

$$\text{basis}^{*} \triangleright \mathit{basis}{:}\mathit{sig}_{\mathit{basis}} \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{ans}$$

and

$$\text{basis}^{*} \triangleright \mathit{basis}{:}\mathit{sig}_{\mathit{basis}} \vdash \mathit{expr} \rightsquigarrow \mathit{exp}' : \mathit{ans}$$

and

$$(\cdot, \cdot, \cdot, [], \mathsf{let}\ \mathit{basis}{=}\mathit{mod}_{\mathit{basis}}\ \mathsf{in}\ \mathit{exp}\ \mathsf{end}) \hookrightarrow^{*} \Sigma_t$$

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

$$(\cdot, \cdot, \cdot, [], \mathsf{let}\ \mathit{basis}{=}\mathit{mod}_{\mathit{basis}}\ \mathsf{in}\ \mathit{exp}'\ \mathsf{end}) \hookrightarrow^{*} \Sigma'_t.$$

Proof (sketch): Non-determinism arises from only three sources. First, the elaborator may "guess" types for bound variables (in `fn` expressions) and when instantiating polymorphism. For example, the EL expression

```
(fn x => 3)(fn y => y)
```

has infinitely many elaborations, including

$$(\lambda\overline{\mathtt{x}}{:}\mathsf{Int}{\rightharpoonup}\mathsf{Int}.3)\,(\lambda\overline{\mathtt{y}}{:}\mathsf{Int}.\overline{\mathtt{y}})$$

and

$$(\lambda\overline{\mathtt{x}}{:}\mathsf{Float}{\rightharpoonup}\mathsf{Float}.3)\,(\lambda\overline{\mathtt{y}}{:}\mathsf{Float}.\overline{\mathtt{y}}).$$

However, the dynamic semantics is sufficiently abstract that choices of types do not affect the course of evaluation, and this indeterminacy cannot cause incoherence.

Second, the elaborator must decide whether valuable expressions should be monomorphic or polymorphic, and in the latter case, on how many type variables to generalize. This turns expressions into functor abstractions, which in theory could cause incoherence by changing the order of evaluation—the expression wrapped by a functor will not be evaluated until the functor is applied. However, we only generalize valuable expressions. Because of the value restriction, if an expression can be generalized it cannot matter when or if the expression is evaluated; observable behavior will be unaffected.

Finally, there is nondeterminism in the creation of equality functions: there may be different defintions of equality at the same type. However, we maintain the invariant that equality functions always perform structural equality on the underlying values regardless of abstraction. Any two candidate definitions then have the same behavior, and no incoherence can arise. ∎

8 Adding Laziness to SML 96

The SML '96 restriction of recursive definitions to syntactic function values makes it very inconvenient to work with recursive lazy (memoized) data structures. As an experiment, we have extended the EL and the translation along the lines suggested by Chris Okasaki [Oka96]:

1. We add the EL type constructor `'a susp` of suspensions (memoized delayed computations) whose value when forced will be of type `'a`.
2. We add the EL expression form \$ *expr*; this creates a suspension which will evaluate *expr* when forced.
3. We allow the forcing of suspensions through the pattern-matching mechanism. A suspension matches the EL pattern \$ *pat* by forcing the suspension, and matching its value against *pat*. This is along the same lines as the implicit dereferencing due to `ref` *pat* patterns.
4. We define equality on suspensions to be equality on their forced values. This makes, for example, equality on finite streams to behave just like equality on lists.

Note that the internal language does not change!

The changes make the use of lazy data structures—especially recursively-defined lazy data structures—much more tractable, but have unusual effects on the language: polymorphic equality may longer being a total function, and effects may occuring in the midst of a pattern-match.

8.1 Syntax Changes

We make the following additions to the EL Syntax:

$$
\begin{array}{rcll}
\mathit{expr} & ::= & \cdots & \\
& \mid & \$\ \mathit{expr} & \text{memoized suspension} \\[1ex]
\mathit{pat} & ::= & \cdots & \\
& \mid & \$\ \mathit{pat} & \text{match value of suspension}
\end{array}
$$

8.2 Basis Changes

We add the following (sealed!) structure to the basis structure *basis* as the substructure "Susp".

$$
\begin{array}{l}
[\mathrm{susp}{\triangleright}t{=}\lambda(u{:}\Omega).u\ \mathsf{Ref} \\
\ \mathrm{delay}{\triangleright}\lambda(s{:}\,[T{\triangleright}\Omega]).\ \big[\mathrm{it}{\triangleright}\lambda(f{:}\mathsf{Unit}{\rightharpoonup}s.T).(\mathsf{ref}^{s.T}\ f)\big] \\
\ \mathrm{force}{\triangleright}\lambda(s{:}\,[T{\triangleright}\Omega]).\ [\mathrm{it}{\triangleright}\lambda(r{:}\mathsf{Unit}{\rightharpoonup}s.T).((\mathsf{get}\,r)\,\{\})] \\
\ \mathrm{update}{\triangleright}\lambda(s{:}\,[T{\triangleright}\Omega]).\ [\mathrm{it}{\triangleright}\lambda(x{:}(s.T\ \mathsf{Ref})\times(\mathsf{Unit}{\rightharpoonup}s.T)).(\mathsf{set}\,(\pi_1\,x,\pi_2\,x))]]\ : \\
[\mathrm{susp}{\triangleright}t{:}\Omega{\Rightarrow}\Omega \\
\ \mathrm{delay}{\triangleright}(s{:}\,[T{\triangleright}\Omega]){\rightharpoonup}\,[\mathrm{it}{\triangleright}(\mathsf{Unit}{\rightharpoonup}s.T){\rightharpoonup}(t\ s.T)] \\
\ \mathrm{force}{\triangleright}(s{:}\,[T{\triangleright}\Omega]){\rightharpoonup}\,[\mathrm{it}{\triangleright}(t\ s.T){\rightharpoonup}s.T] \\
\ \mathrm{update}{\triangleright}(s{:}\,[T{\triangleright}\Omega]){\rightharpoonup}\,[\mathrm{it}{\triangleright}((t\ s.T)\times(\mathsf{Unit}{\rightharpoonup}s.T)){\rightharpoonup}\mathsf{Unit}]]
\end{array}
$$

8.3 Elaboration Changes

We add the following rules to the elaboration section:

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : con}{\Gamma \vdash \$\ expr \rightsquigarrow ((basis.\mathrm{Susp.delay}\,[T \triangleright con]).\mathrm{it})\; exp : (basis.\mathrm{Susp.susp}\; con)}$$

Creation of a single memoized suspension.

$$\frac{\begin{array}{c} lbl_1, \cdots, lbl_n \notin Rng\,\overline{\cdot} \qquad var_1, \cdots, var_n \notin \mathrm{BV}(\Gamma) \\ \Gamma' = \Gamma, \overline{id_1} \triangleright var_1{:}(basis.\mathrm{Susp.susp}\; con_1), \cdots, \overline{id_n} \triangleright var_n{:}(basis.\mathrm{Susp.susp}\; con_n) \\ \Gamma' \vdash expr_1 \rightsquigarrow exp_1 : con_1 \quad \cdots \quad \Gamma' \vdash expr_n \rightsquigarrow exp_n : con_n \end{array}}{\begin{array}{l} \Gamma \vdash\ \texttt{val rec}\ \{(lab_i\texttt{=}id_i)_{i=1}^n\}\ \texttt{=}\ \{(lab_i\texttt{=}\$\ expr_i)_{i=1}^n\} \hookrightarrow \\ \qquad [\overline{id_1} \triangleright var_1 {=} ((basis.\mathrm{Susp.delay}\,[T \triangleright con_1]).\mathrm{it})\,(\lambda\{\}.\mathsf{fail}^{con_1}), \cdots, \\ \qquad \overline{id_n} \triangleright var_n {=} ((basis.\mathrm{Susp.delay}\,[T \triangleright con_n]).\mathrm{it})\,(\lambda\{\}.\mathsf{fail}^{con_n}), \\ \qquad lbl_1 \triangleright ((basis.\mathrm{Susp.update}\,[T \triangleright var_1]).\mathrm{it})\,(\lambda\{\}.exp_1), \cdots, \\ \qquad lbl_n \triangleright ((basis.\mathrm{Susp.update}\,[T \triangleright var_n]).\mathrm{it})\,(\lambda\{\}.exp_1)] : \\ \qquad [\overline{id_1} \triangleright var_1{:}(basis.\mathrm{Susp.susp}\; con_1), \cdots, \\ \qquad \overline{id_n} \triangleright var_n{:}(basis.\mathrm{Susp.susp}\; con_n), \\ \qquad lbl_1 \triangleright \mathsf{Unit}, \cdots, lbl_n \triangleright \mathsf{Unit}] : \end{array}}$$

Binding of recursive suspensions. Note that there is no polymorphism possible here: the creation of a suspension involves the application of a partial function. (In particular, a new reference cell is allocated when a suspension is created.)

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \Omega}{\Gamma \vdash ty\ \texttt{susp} \rightsquigarrow (basis.\mathrm{Susp.susp}\; con) : \Omega}$$

Translation of suspension types.

$$\frac{\Gamma \vdash pat \Leftarrow \mathsf{get}\; exp : con \mid exp' \rightsquigarrow mod : sig}{\Gamma \vdash \texttt{ref}\ pat \Leftarrow exp : con\;\mathsf{Ref} \mid exp' \rightsquigarrow mod : sig}$$

Suspensions are forced via pattern-matching. Since pattern-matching can now have side-effects, it becomes important to note that our reference pattern-compiler checks values against patterns in depth-first left-to-right order, and that the process short-circuits as soon as a mismatch is found. Therefore the function

```
fun f(true, $ ()) = 0
  | f(false, _)  = 1
```

applied to `(false,$ (print "hello"))` will return `1` but not print hello.

$$\frac{\begin{array}{c} decs \vdash con \equiv (basis.\mathrm{Susp.susp}\; con') : \Omega \\ decs \vdash_{\mathrm{eq}} con' \rightsquigarrow exp_{\mathrm{v}} \end{array}}{\begin{array}{l} decs \vdash_{\mathrm{eq}} con \rightsquigarrow \\ \qquad \lambda var{:}con \times con.exp_{\mathrm{v}}\,(((basis.\mathrm{Susp.force}\,[T \triangleright con]).\mathrm{it})\,(\pi_1\, var), \\ \qquad\qquad ((basis.\mathrm{Susp.force}\,[T \triangleright con]).\mathrm{it})\,(\pi_2\, var)) \end{array}}$$

Suspensions are compared for equality not by equality of thunks, but by equality of their values, to preserve the analogy between lists and (finite) lazily-evaluated streams. Note that this could cause polymorphic equality to have side-effects or be nonterminating.

9 Acknowledgments

We would like to thank Perry Cheng for helping us to find and correct errors in the work presented here.

References

[GR] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library reference manual.* Forthcoming.

[Har93] Robert Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, School of Computer Science, Carnegie Mellon University, 1993.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[HM93] 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.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*, pages 130–141, 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, pages 409–423. Springer-Verlag, LNCS 788, April 1994.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MTHM96] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML, Revised 1996.* 1996. Forthcoming.

[Oka96] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Aarhus University, September 1981.

[Tar96] David Tarditi. *Optimizing ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

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

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

[WF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, 1991.