

A Type-Theoretic Account of Standard ML 1996 (Version 1)

Chris Stone Robert Harper

May 10, 1996

CMU-CS-96-136

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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 translation of SML96 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 and signature matching) into uses of the more rudimentary mechanisms of the intermediate language.

This document presents work in progress and is being distributed for the purpose of obtaining feedback. As such, the translation does not completely match the definition of SML96, which is itself still undergoing change.

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

Contents

1	Overview	3
2	Major Differences from SML96	6
2.1	Equality	6
2.2	Value Restriction	6
2.3	Type Sharing and Datatypes	7
2.4	Local and Higher-Order Functors	8
2.5	Top-level	8
2.6	Principality	8
2.7	Overloading	8
3	Internal Language Abstract Syntax	9
3.1	Core Expressions	9
3.2	Core Constructors and Kinds	10
3.3	Module Expressions	11
3.4	Module Signatures	12
3.5	Top-level	13
3.6	Translation Context	14
3.7	Syntactic Restrictions	14
3.8	Notation	14
3.9	Bindings and Scope	15
4	Static Semantics: Typing	17
4.1	Introduction	17
4.2	Notation	17
4.3	Judgment Forms	18
4.4	Inference Rules	19
5	Static Semantics: Valuability	33
5.1	Judgment Forms	33
5.2	Syntactic Values	34
5.3	Inference Rules	34
6	Dynamic Semantics	37
6.1	Introduction	37
6.2	Frames	38
6.3	Raise Frames	39

6.4	Judgment Forms	40
6.5	Inference Rules	40
7	External Language	47
7.1	Notation	47
7.2	Grammar of the Abstract Syntax	47
7.3	Syntactic Restrictions	50
8	The Translation	51
8.1	Introduction	51
8.2	Notation	52
8.3	Initial Basis	53
8.4	Judgment Forms	54
8.5	Inference Rules	54
8.6	Pattern Compilation	71
8.7	Lookup Rules	74
8.8	Coercions	78
8.9	Signature Patching	82
9	Conjectures	84

1 Overview

This document consists of a type-theoretic account of a variant of Standard ML (Revised 1996). The approach taken here is to define the static and dynamic semantics of SML96 by a syntax-directed translation of the SML96 abstract syntax (the *external language (EL)*) into an explicitly-typed λ -calculus (the *internal language (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 used in *The Definition of Standard ML* [MTH90], which gives a dynamic semantics directly to the “raw” 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 SML96 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 SML96. 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 SML96. 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 SML96 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 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

$$\Gamma \vdash EL\text{-}phrase \rightsquigarrow IL\text{-}phrase : IL\text{-}classifier$$

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

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

also to hold.

2 Major Differences from SML96

This section refers only to the draft of Standard ML '96 as of the time of this report. Many aspects of SML96 are still under consideration, and may change substantially!

2.1 Equality

This translation does not currently handle equality types, the **eqtype** signature specification, or polymorphic equality. We may relax this restriction (in a later draft of this translation) to allow **eqtype** declarations and the structural equality derivable from these and known types. We do not intend to include equality polymorphism (**'a**), which significantly complicates both the semantics and the implementation of the language.

2.2 Value Restriction

The value restriction used in this translation is different than that currently specified in SML96.

- We treat

$$\text{val } \langle \text{rec} \rangle \text{ } pat_1 = expr_1 \text{ and } \dots \text{ and } pat_n = expr_n$$

as a derived form for

$$\text{val } \langle \text{rec} \rangle (pat_1, \dots, pat_n) = (exp_1, \dots, exp_n).$$

If any of $exp_1, \dots, exp_n$ is not a 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** $h = \text{hd } v$ and **val** $t = \text{tl } v$. 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 matches the notion of “polymorphism as substitution,” in which `let val id=exp in exp' end` is considered equivalent to $[exp/id]exp'$ when *id* is polymorphic.

- We distinguish constructors from user-defined functions by their types: constructors can have total function types and `fn` expressions should express partial types. However, the translation can sometimes notice that certain user variables and functions are total, and assigns them total types in the translated code. For example,

```
val x = op ::
val y = x (nil, nil)
```

can give `x` the same total type as the `::` constructor, so `y` can be polymorphic (since a total function applied to a value is valuable and hence generalizable.)

Thus we allow certain (sound) programs than SML96 would reject; if we added the ability to express total types in the EL, this might even be acceptable. Alternatively, we will have to modify the translation so that it “forgets” that certain values are total functions.

2.3 Type Sharing and Datatypes

The translation allows certain unsatisfiable constraints to appear in EL signatures. In particular, a type sharing constraint equating two “obviously different” datatypes (datatypes with different sets of constructors) will not be flagged. The specification will elaborate, but there will be no structures in the range of the translation that match the specification.

In the language of the *Definition*, we are relaxing the consistency requirement.

2.4 Local and Higher-Order Functors

Our external language permits structure and functor declarations within a `let` or `local` declarations—a reflection of the fact that the internal language has local modules.

2.5 Top-level

We are interested in batch compilers, which may compile top-level declarations separately. Therefore, we require that:

1. Only structures, functors, and signatures may be defined at top-level.
2. All identifiers bound at the top level must be distinct (taking into account the fact that structure, functor, and signature identifier namespaces are always considered disjoint) so that a linker can resolve references correctly.

2.6 Principality

The translation makes no requirement that top-level declarations, or even entire compilation units, have principal environments. A principality requirement may be useful for efficient implementation, and implementations may choose to make such restrictions, but this is purely the decision of the implementor. For example, an implementation could instead delay some typechecking and/or compiling until the entire program becomes known.

2.7 Overloading

This translation does not consider overloaded operators. It may be sufficient to add an “overloaded” type and add a rule to the translation of expressions such that identifiers with overloaded types are coerced to a non-overloaded type whenever it appears. (This would be similar to the current rule that instantiates polymorphic identifiers to monomorphic expressions wherever they appear.)

3 Internal Language Abstract Syntax

3.1 Core Expressions

$path ::=$	$var \mid var.lbls$	
$lbls ::=$	$lbl \mid lbl.lbls$	
$exp ::=$	$scon$	(constants)
	$ var$	(variables)
	$ loc$	(memory locations)
	$ name$	(tag names for the Any type)
	$ exp \ exp'$	(application)
	$ \text{fix } fbnds \text{ in } var \text{ end}$	(projection from fixpoint)
	$ \{rbnds\}$	(record of values)
	$ exp \# lbl$	(projection from a record)
	$ \text{handle } exp \text{ with } exp'$	(handle exceptions at type Any)
	$ \text{raise } exp$	(raise exceptions at type Any)
	$ \text{catch } exp \text{ with } exp'$	(handle exceptions at type Unit)
	$ \text{fail}$	(raise exceptions at type Any)
	$ \text{let } var = mod \text{ in } exp \text{ end}$	(local module definition)
	$ \text{new_stamp}[con]$	(new constructor/deconstructor pair)
	$ \text{ref}^{con} \ exp$	(allocate new ref cell)
	$ \text{get } exp$	(dereference)
	$ \text{set } exp$	(assignment)
	$ \text{roll}_i^{con} \ exp$	(type coercion into a recursive type)
	$ \text{unroll}_i^{con} \ exp$	(type coercion out of a recursive type)
	$ \text{inj}_i^{con} \ exp$	(injection into sum type)
	$ \text{proj}_i \ exp$	(projection from sum type)
	$ \text{tag}^{name} \ exp$	(injection into Any type)
	$ \text{untag}^{name} \ exp$	(projection from Any type)
	$ \text{case}^{con} (exp_1, \dots, exp_n) \text{ of } exp \text{ end}$	(case over sum type)
	$ mod.lbl$	(projection from module)
$rbnds ::=$	$\cdot \mid rbnds, rbnd$	
$rbnd ::=$	$lbl \triangleright exp$	
$fbnds ::=$	$\cdot \mid fbnds, fbnd$	
$fbnd ::=$	$var' = (var : con) \mapsto exp$	

We use “ $\lambda var : con. exp$ ” as an abbreviation for “ $\text{fix } var' = (var : con) \mapsto exp \text{ in } var' \text{ end}$ ” where var' does not occur free in exp .

3.2 Core Constructors and Kinds

$con ::=$	var	(type variables)
	$ \text{Int} \mid \text{Float} \mid \text{String} \mid \text{Char} \mid \dots$	(base types)
	$ \text{Any}$	(extensible sum type)
	$ con \text{ Ref}$	(reference type)
	$ con \text{ Name}$	(tag-name type)
	$ con \multimap con'$	(partial function type)
	$ con \rightarrow con'$	(total function type)
	$ con [con']$	(application)
	$ \mu_i [con]$	(projection from fixpoint)
	$ \text{Unroll} [con]$	(unroll recursive type)
	$ \{rdec\}$	(record type)
	$ \lambda (var_1, \dots, var_n).con \ (n \geq 0)$	(constructor-level function)
	$ \Sigma (con_1, \dots, con_n)$	(sum type)
	$ (con_1, \dots, con_n)$	(tuple formation)
	$ con \# k \ (k \geq 1)$	(tuple projection)
	$ mod.lbl$	(module projection)
$rdec ::=$	$\cdot \mid rdec, rdec$	
$rdec ::=$	$lbl \triangleright con$	
$knd ::=$	$\Omega^n \quad (n \geq 0)$	(tuples of types)
	$ \Omega^m \Rightarrow \Omega^n \ (m, n \geq 0)$	

We refer to the constructor $\{\}$ as Unit.

3.3 Module Expressions

$mod ::=$	var	
	$ [sbnds]$	(structure)
	$ (\lambda var: sig.mod)$	(functor)
	$ mod\ mod'$	(functor application)
	$ mod.lbl$	(projection from module)
	$ mod:sig$	(forgetting type abbreviations)
$sbnds ::=$	$\cdot$	(structure bindings)
	$ sbnds, sbnd$	
$sbnd ::=$	$lbl \triangleright bnd$	
$bnd ::=$	$var = exp$	
	$ var = mod$	
	$ var = con$	

For readability, we often elide the internal names (*var*'s) when writing out *sbnds* (and *sdec*s). In all cases it should be immediately obvious how to consistently restore variables not appearing anywhere in the inference rule.

3.4 Module Signatures

$sig ::=$	$[sdecs]$	
	$ ((var : sig) \multimap sig')$	(partial functor signature)
	$ ((var : sig) \rightarrow sig')$	(total functor signature)
$sdecs ::=$	$\cdot$	(structure declarations)
	$ sdecs, sdec$	
$sdec ::=$	$lbl \triangleright dec$	
$decs ::=$	$\cdot$	(declaration lists)
	$ decs, dec$	
$dec ::=$	$var : con$	
	$ var : sig$	
	$ var : kind$	(opaque type declaration)
	$ var : kind = con$	(type abbreviation)
	$ loc : con$	(typed locations)
	$ name : con$	(typed exception names)

We use $(sig \multimap sig')$ and $(sig \rightarrow sig')$ to abbreviate $((var : sig) \multimap sig')$ and $((var : sig) \rightarrow sig')$ respectively, where var is not free in sig' .

3.5 Top-level

$$\begin{aligned} tdecs &::= \cdot \\ &\quad | \quad tdecs, tdec \\ tdec &::= \textit{lbl} \triangleright \textit{var} : \textit{sig} \\ &\quad | \quad \textit{lbl} \triangleright \textit{var} : \textbf{Sig} = \textit{sig} \\ tbnds &::= \cdot \\ &\quad | \quad tbnds, tbnd \\ tbnd &::= \textit{lbl} \triangleright \textit{var} = \textit{mod} \\ &\quad | \quad \textit{lbl} \triangleright \textit{var} = \textit{sig} \end{aligned}$$

3.6 Translation Context

$$\begin{aligned} \Gamma ::= & \cdot \\ & | \Gamma, lbl \triangleright var : con \\ & | \Gamma, lbl \triangleright var : kind \langle = con \rangle \\ & | \Gamma, lbl \triangleright var : sig \\ & | \Gamma, lbl \triangleright var : Sig = sig \end{aligned}$$

There is an implicit coercion of a translation context (Γ) to a sequence of declarations ($decs$), induced by dropping all signature bindings and forgetting all lbl 's.

3.7 Syntactic Restrictions

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

3.8 Notation

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

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

with analogous definitions for all the other classes listed above.

- We occasionally use the abbreviation

$$(phrase_i)_{i=1}^n$$

as shorthand for

$$phrase_1, \dots, phrase_n$$

(where the phrases are comma-separated).

3.9 Bindings and Scope

We define the functions $\text{bound}(\cdot)$ and $\text{dom}(\cdot)$ for various declarations (and lists thereof):

<i>Function</i>	<i>Definition</i>
$\text{bound}(dec)$	$\text{bound}(var:con) = var$ $\text{bound}(var:knd) = var$ $\text{bound}(var:knd=con) = var$ $\text{bound}(var:sig) = var$ $\text{bound}(loc:con) = loc$ $\text{bound}(name:con) = name$
$\text{bound}(decs)$	$\text{bound}(dec_1, \dots, dec_n) = \{\text{bound}(dec_1), \dots, \text{bound}(dec_n)\}$
$\text{bound}(sdecs)$	$\text{dom}(lbl_1 \triangleright dec_1, \dots, lbl_n \triangleright dec_n) = \{\text{bound}(dec_1), \dots, \text{bound}(dec_n)\}$
$\text{bound}(bnd)$	$\text{bound}(var=exp) = var$ $\text{bound}(var=con) = var$ $\text{bound}(var=mod) = var$
$\text{bound}(sbnds)$	$\text{dom}(lbl_1 \triangleright bnd_1, \dots, lbl_n \triangleright bnd_n) = \{\text{bound}(bnd_1), \dots, \text{bound}(bnd_n)\}$
$\text{bound}(\Gamma)$	$\text{bound}(sdec, \Gamma) = \{\text{bound}(sdec)\} \cup \text{dom}(\Gamma)$ $\text{bound}(lbl \triangleright var:\text{Sig}=sig, \Gamma) = \{var\} \cup \text{dom}(\Gamma)$
$\text{dom}(sdecs)$	$\text{dom}(lbl_1 \triangleright dec_1, \dots, lbl_n \triangleright dec_n) = \{lbl_1, \dots, lbl_n\}$
$\text{dom}(rdecs)$	$\text{dom}(lbl_1 \triangleright con_1, \dots, lbl_n \triangleright con_n) = \{lbl_1, \dots, lbl_n\}$
$\text{dom}(tdec)$	$\text{dom}(lbl \triangleright var:sig) = lbl$ $\text{dom}(lbl \triangleright var:\text{Sig}=sig) = lbl$
$\text{dom}(tdecs)$	$\text{dom}(tdec_1, \dots, tdec_n) = \{\text{dom}(tdec_1), \dots, \text{dom}(tdec_n)\}$
$\text{dom}(\Gamma)$	$\text{dom}(sdec, \Gamma) = \{\text{dom}(sdec)\} \cup \text{dom}(\Gamma)$ $\text{dom}(lbl \triangleright var:\text{Sig}=sig, \Gamma) = \{lbl\} \cup \text{dom}(\Gamma)$

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

<i>Binding Phrase</i>	<i>Bound</i>	<i>Scope</i>
$\text{fix } fbnds, var' = (var:con) \mapsto exp, fbnds' \text{ in } var \text{ end}$ $\text{let } var = mod \text{ in } exp \text{ end}$	var' var var	entire phrase exp exp
$\lambda (var_1, \dots, var_n).con$	$var_1, \dots, var_n$	con
$s bnd, sbnds$ $(\lambda var: sig.mod)$	$\text{bound}(s bnd)$ var	$sbnds$ mod
$sdec, sdec s$ $(var: sig \rightarrow sig')$ $(var: sig \rightarrow sig')$	$\text{bound}(sdec)$ var var	$sdec s$ sig' sig'
$t bnd, tbnds$ $tdec, tdec s$	$\text{bound}(t bnd)$ $\text{bound}(tdec)$	$tbnds$ $tdec s$
$sdec, \Gamma$ $lbl \triangleright var: Sig = sig, \Gamma$	$\text{bound}(sdec)$ 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 $FV(phrase)$ to denote the set of free expression variables in *phrase*, and $FTV(phrase)$ to denote the set of free type variables in the phrase.

4 Static Semantics: Typing

4.1 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.
- There is a mutual dependency between the typing judgments for (core and module) expressions and the valuability judgments of Section 5.

4.2 Notation

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

4.3 Judgment Forms

<i>Judgment...</i>	<i>Meaning...</i>
$\vdash decs \text{ ok}$	$decs$ is a valid declaration list
$\vdash decs \leq decs'$	$decs$ is a sub-declaration list of $decs'$
$\vdash decs \equiv decs'$	$decs$ and $decs'$ are equivalent declaration lists
$decs \vdash dec \text{ ok}$	dec is a valid declaration
$decs \vdash dec \leq dec'$	dec is a sub-declaration of dec'
$decs \vdash dec \equiv dec'$	dec and dec' are equivalent declaration lists
$decs \vdash bnd : dec$	binding bnd has declaration dec
$decs \vdash rdecs \text{ ok}$	$rdecs$ is a valid record declaration list
$decs \vdash rdecs \equiv rdecs'$	$rdecs$ and $rdecs'$ are equivalent record declaration lists
$decs \vdash rbnds : rdecs$	record binding list $rbnds$ has declaration list $rdecs$
$decs \vdash kind : \mathbf{Kind}$	$kind$ is a valid kind
$decs \vdash con : kind$	con has kind $kind$
$decs \vdash con \equiv con' : kind$	con and con' are equivalent constructors of kind $kind$
$decs \vdash exp : con$	expression exp has type con
$decs \vdash sdecs \text{ ok}$	$sdecs$ is a valid signature specification list
$decs \vdash sdecs \leq sdecs'$	$sdecs$ is a sub-specification list of $sdecs'$
$decs \vdash sdecs \equiv sdecs'$	$sdecs$ and $sdecs'$ are equivalent specification lists
$decs \vdash sbnds : sdecs$	function binding list $sbnds$ has declaration list $sdecs$
$decs \vdash sig : \mathbf{Sig}$	sig is a valid signature
$decs \vdash sig \leq sig' : \mathbf{Sig}$	sig is a sub-signature of sig'
$decs \vdash sig \equiv sig' : \mathbf{Sig}$	sig and sig' are equivalent signatures
$decs \vdash mod : sig$	mod has signature sig
$decs \vdash tdecs \text{ ok}$	$tdecs$ is a valid top-level declaration list
$decs \vdash tbnds : tdecs$	top-level bindings $tbnds$ have declarations $tdecs$

4.4 Inference Rules

$$\boxed{\vdash decs \text{ ok}}$$

$$\frac{}{\vdash \cdot \text{ ok}} \quad (1)$$

$$\frac{decs \vdash dec \text{ ok}}{\vdash decs, dec \text{ ok}} \quad (2)$$

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

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

$$\frac{}{\vdash \cdot \leq \cdot} \quad (3)$$

$$\frac{\vdash decs \leq decs' \quad decs \vdash dec \leq dec'}{\vdash decs, dec \leq decs', dec'} \quad (4)$$

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

$$\frac{}{\vdash \cdot \equiv \cdot} \quad (5)$$

$$\frac{\vdash decs \equiv decs' \quad decs \vdash dec \equiv dec'}{\vdash decs, dec \equiv decs', dec'} \quad (6)$$

$$\boxed{decs \vdash dec \text{ ok}}$$

$$\frac{decs \vdash knl : \text{Kind} \quad var \notin \text{bound}(decs)}{decs \vdash var:knl \text{ ok}} \quad (7)$$

$$\frac{decs \vdash con : knl \quad var \notin \text{bound}(decs)}{decs \vdash var:knl=con \text{ ok}} \quad (8)$$

$$\frac{decs \vdash con : \Omega \quad var \notin \text{bound}(decs)}{decs \vdash var:con \text{ ok}} \quad (9)$$

$$\frac{decs \vdash sig : \text{Sig} \quad var \notin \text{bound}(decs)}{decs \vdash var:sig \text{ ok}} \quad (10)$$

$$\frac{decs \vdash con : \Omega \quad loc \notin \text{bound}(decs)}{decs \vdash loc:con \text{ Ref ok}} \quad (11)$$

$$\frac{decs \vdash con : \Omega \quad name \notin \text{bound}(decs)}{decs \vdash name:con \text{ Name ok}} \quad (12)$$

$$\boxed{decs \vdash dec \leq dec'}$$

$$\frac{decs \vdash con : kind}{decs \vdash var:kind=con \leq var:kind} \quad (13)$$

$$\frac{decs \vdash sig \leq sig' : \text{Sig}}{decs \vdash var:sig \leq var:sig'} \quad (14)$$

$$\frac{decs \vdash dec \equiv dec'}{decs \vdash dec \leq dec'} \quad (15)$$

$$\boxed{decs \vdash dec \equiv dec'}$$

$$\frac{decs \vdash kind : \text{Kind}}{decs \vdash var:kind \equiv var:kind} \quad (16)$$

$$\frac{decs \vdash con \equiv con' : kind}{decs \vdash var:kind=con \equiv var:kind=con'} \quad (17)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash var:con \equiv var:con'} \quad (18)$$

$$\frac{decs \vdash sig \equiv sig' : \text{Sig}}{decs \vdash var:sig \equiv var:sig'} \quad (19)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash loc:con \equiv loc:con'} \quad (20)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash name:con \equiv name:con'} \quad (21)$$

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

$$\frac{decs \vdash con : knđ}{decs \vdash var=con : var:knđ} \quad (22)$$

$$\frac{decs \vdash con \equiv con' : knđ}{decs \vdash var=con : var:knđ=con'} \quad (23)$$

$$\frac{decs \vdash exp : con}{decs \vdash var=exp : var:con} \quad (24)$$

$$\frac{decs \vdash mod : sig \quad decs \vdash sig' \leq sig : Sig}{decs \vdash var=mod : var:sig} \quad (25)$$

$$\boxed{decs \vdash rdecs \text{ ok}}$$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot \text{ ok}} \quad (26)$$

$$\frac{\begin{array}{l} decs \vdash rdecs \text{ ok} \\ decs \vdash con : \Omega \\ lbl \notin \text{dom}(rdecs) \end{array}}{decs \vdash rdecs, lbl \triangleright con \text{ ok}} \quad (27)$$

$$\boxed{decs \vdash rdecs \equiv rdecs'}$$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot \equiv \cdot} \quad (28)$$

$$\frac{decs \vdash rdecs \equiv rdecs' \quad decs \vdash con \equiv con' : \Omega}{decs \vdash rdecs, lbl \triangleright con \equiv rdecs', lbl \triangleright con'} \quad (29)$$

$$\frac{decs \vdash rdecs, lbl \triangleright con, lbl' \triangleright con', rdecs' \text{ ok}}{decs \vdash rdecs, lbl \triangleright con, lbl' \triangleright con', rdecs' \equiv rdecs, lbl' \triangleright con', lbl \triangleright con, rdecs'} \quad (30)$$

$$\boxed{decs \vdash rbnds : rdecs}$$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot : \cdot} \quad (31)$$

$$\frac{decs \vdash rbnds : rdecs \quad decs \vdash exp : con}{decs \vdash rbnds, lbl \triangleright exp : rdecs, lbl \triangleright con} \quad (32)$$

$$\boxed{decs \vdash knnd : \mathbf{Kind}}$$

$$\frac{n \geq 0}{decs \vdash \Omega^n : \mathbf{Kind}} \quad (33)$$

$$\frac{n, m \geq 0}{decs \vdash \Omega^m \Rightarrow \Omega^n : \mathbf{Kind}} \quad (34)$$

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

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var : knnd \langle = con \rangle, decs''}{decs \vdash var : knnd} \quad (35)$$

$$\overline{decs \vdash \mathbf{Any} : \Omega} \quad (36)$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \mathbf{Ref} : \Omega} \quad (37)$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \mathbf{Name} : \Omega} \quad (38)$$

$$\frac{decs \vdash con : \Omega \quad decs \vdash con' : \Omega}{decs \vdash con \multimap con' : \Omega} \quad (39)$$

$$\frac{decs \vdash con : \Omega \quad decs \vdash con' : \Omega}{decs \vdash con \rightarrow con' : \Omega} \quad (40)$$

$$\frac{decs \vdash rdecs \text{ ok}}{decs \vdash \{rdecs\} : \Omega} \quad (41)$$

$$\frac{decs \vdash con : \Omega^n \Rightarrow \Omega^n \quad 1 \leq i \leq n}{decs \vdash \mu_i [con] : \Omega} \quad (42)$$

$$\frac{decs \vdash con \equiv \mu_i [con'] : \Omega}{decs \vdash \text{Unroll} [con] : \Omega} \quad (43)$$

$$\frac{decs, var_1:\Omega, \dots, var_n:\Omega \vdash con : \Omega^m}{decs \vdash \lambda(var_1, \dots, var_n).con : \Omega^n \Rightarrow \Omega^m} \quad (44)$$

$$\frac{decs \vdash con_1 : \Omega \quad \dots \quad decs \vdash con_n : \Omega}{decs \vdash \Sigma(con_1, \dots, con_n) : \Omega} \quad (45)$$

$$\frac{decs \vdash con_1 : \Omega \quad \dots \quad decs \vdash con_n : \Omega}{decs \vdash (con_1, \dots, con_n) : \Omega^n} \quad (46)$$

$$\frac{decs \vdash con : con' \Rightarrow con \quad decs \vdash con : kend'}{decs \vdash con [con'] : kend} \quad (47)$$

$$\frac{decs \vdash mod : [sdecs, lbl \triangleright var:kend \langle =con \rangle, sdecs']}{decs \vdash mod.lbl : kend} \quad (48)$$

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

$$\frac{decs \vdash con \equiv con' : kend \quad decs = decs', var:kend=con', decs''}{decs \vdash var \equiv con : kend} \quad (49)$$

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \dots, lbl_m \triangleright dec_m, lbl \triangleright var:kend=con', sdecs] \\ decs, dec_1, \dots, dec_m \vdash con \equiv con' : kend \\ decs \vdash con : kend \end{array}}{decs \vdash mod.lbl \equiv con : kend} \quad (50)$$

Rule 50: The projection must be a valid constructor with respect to the ambient context.

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \quad decs \vdash con'_1 \equiv con'_2 : \Omega}{decs \vdash con_1 \rightarrow con'_1 \equiv con_2 \rightarrow con'_2 : \Omega} \quad (51)$$

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \quad decs \vdash con'_1 \equiv con'_2 : \Omega}{decs \vdash con_1 \rightarrow con'_1 \equiv con_2 \rightarrow con'_2 : \Omega} \quad (52)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con \text{ Ref} \equiv con' \text{ Ref} : \Omega} \quad (53)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con \mathbf{Name} \equiv con' \mathbf{Name} : \Omega} \quad (54)$$

$$\frac{decs \vdash rdecs \equiv rdecs'}{decs \vdash \{rdecs\} \equiv \{rdecs'\} : \Omega} \quad (55)$$

$$\frac{decs \vdash con \equiv con' : \Omega^n \Rightarrow \Omega^n \quad (1 \leq i \leq n)}{decs \vdash \mu_i[con] \equiv \mu_i[con'] : \Omega} \quad (56)$$

$$\frac{decs \vdash con \equiv \mu_i[con'] : \Omega \quad decs \vdash con' : \Omega^n \Rightarrow \Omega^n}{decs \vdash \mathbf{Unroll}[con] \equiv (con'[(\mu_1[con'], \dots, \mu_n[con'])])\#i : \Omega} \quad (57)$$

$$\frac{decs, var_1:\Omega, \dots, var_n:\Omega \vdash con \equiv con' : \Omega}{decs \vdash \lambda(var_1, \dots, var_n).con \equiv \lambda(var_1, \dots, var_n).con' : \Omega^n \Rightarrow \Omega} \quad (58)$$

$$\frac{decs \vdash con_1 \equiv con'_1 : \Omega \quad \dots \quad decs \vdash con_n \equiv con'_n : \Omega}{decs \vdash \Sigma(con_1, \dots, con_n) \equiv \Sigma(con'_1, \dots, con'_n) : \Omega} \quad (59)$$

$$\frac{decs \vdash con_1 \equiv con'_1 : \Omega \quad \dots \quad decs \vdash con_n \equiv con'_n : \Omega}{decs \vdash (con_1, \dots, con_n) \equiv (con'_1, \dots, con'_n) : \Omega^n} \quad (60)$$

$$\frac{decs \vdash con_1 \equiv con'_1 : kend' \Rightarrow kend \quad decs \vdash con_2 \equiv con'_2 : kend'}{decs \vdash con_1[con_2] \equiv con'_1[con'_2] : kend} \quad (61)$$

$$\frac{\begin{array}{c} decs, var_1:\Omega, \dots, var_n:\Omega \vdash con : \Omega^m \\ decs \vdash con' \equiv (con'_1, \dots, con'_n) : \Omega^n \end{array}}{decs \vdash (\lambda(var_1, \dots, var_n).con)[con'] \equiv [con'_1/var_1] \dots [con'_n/var_n]con : \Omega^m} \quad (62)$$

$$\frac{decs \vdash con : \Omega^n \Rightarrow \Omega^m}{decs \vdash \lambda(var_1, \dots, var_n).(con[(var_1, \dots, var_n)]) \equiv con : \Omega^m} \quad (63)$$

$$\frac{decs \vdash con \equiv (con_1, \dots, con_n) : \Omega^n \quad (1 \leq i \leq n)}{decs \vdash con\#i \equiv con_i : \Omega} \quad (64)$$

$$\frac{decs \vdash con : \Omega^n}{decs \vdash con \equiv (con\#1, \dots, con\#n) : \Omega^n} \quad (65)$$

$$\frac{decs \vdash con : kn d}{decs \vdash con \equiv con : kn d} \quad (66)$$

$$\frac{decs \vdash con' \equiv con : kn d}{decs \vdash con \equiv con' : kn d} \quad (67)$$

$$\frac{decs \vdash con \equiv con' : kn d \quad decs \vdash con' \equiv con'' : kn d}{decs \vdash con \equiv con'' : kn d} \quad (68)$$

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

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var:con, decs''}{decs \vdash var : con} \quad (69)$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', loc:con, decs''}{decs \vdash loc : con} \quad (70)$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', name:con, decs''}{decs \vdash name : con} \quad (71)$$

$$\frac{decs \vdash exp : con' \rightarrow con \quad decs \vdash exp' : con'}{decs \vdash exp \ exp' : con} \quad (72)$$

$$\frac{decs \vdash exp : con' \rightarrow con \quad decs \vdash exp' : con'}{decs \vdash exp \ exp' : con} \quad (73)$$

$$\frac{decs, (var'_i:con_i \rightarrow con'_i)_{i=1}^n, var_k:con_k \vdash exp_k : con'_k \quad (1 \leq k \leq n)}{decs \vdash \text{fix} (var'_i=(var_i:con_i) \mapsto exp_i)_{i=1}^n \text{ in } var'_k \text{ end} : con_k \rightarrow con'_k} \quad (74)$$

$$\frac{\begin{array}{c} var'_1, \dots, var'_n \notin \text{bound}(decs) \\ decs, var_k:con_k \vdash exp_k \downarrow con'_k \end{array}}{decs \vdash \text{fix} (var'_i=(var_i:con_i) \mapsto exp_i)_{i=1}^n \text{ in } var'_k \text{ end} : con_k \rightarrow con'_k} \quad (75)$$

Rule 75: This rule is actually overly restricts the functions that can be given total types, but it suffices.

$$\frac{decs \vdash rbnds : rdecs}{decs \vdash \{rbnds\} : \{rdecs\}} \quad (76)$$

$$\frac{decs \vdash exp : \{rdecs, lbl \triangleright con, rdecs'\}}{decs \vdash exp \# lbl : con} \quad (77)$$

$$\frac{decs \vdash exp : con \quad decs \vdash exp' : \mathbf{Any} \multimap con}{decs \vdash \mathbf{handle} \ exp \ \mathbf{with} \ exp' : con} \quad (78)$$

$$\frac{decs \vdash exp : \mathbf{Any}}{decs \vdash \mathbf{raise} \ exp : con} \quad (79)$$

$$\frac{decs \vdash exp : con \quad decs \vdash exp' : con}{decs \vdash \mathbf{catch} \ exp \ \mathbf{with} \ exp' : con} \quad (80)$$

$$\frac{}{decs \vdash \mathbf{fail} : con} \quad (81)$$

$$\frac{decs \vdash mod : sig \quad decs, var:sig \vdash exp : con \quad decs \vdash con : \Omega}{decs \vdash \mathbf{let} \ var = mod \ \mathbf{in} \ exp \ \mathbf{end} : con} \quad (82)$$

Rule 82: The result type may not depend on abstract types from the local module.

$$\frac{}{decs \vdash \mathbf{new_stamp}[con] : \{\mathbf{mk}:con \multimap \mathbf{Any}, \mathbf{km}:\mathbf{Any} \multimap con\}} \quad (83)$$

$$\frac{decs \vdash exp : con}{decs \vdash \mathbf{ref}^{con} \ exp : con \ \mathbf{Ref}} \quad (84)$$

$$\frac{decs \vdash exp : con \ \mathbf{Ref}}{decs \vdash \mathbf{get} \ exp : con} \quad (85)$$

$$\frac{decs \vdash exp : \{1 \triangleright con \ \mathbf{Ref}, 2 \triangleright con\}}{decs \vdash \mathbf{set} \ exp : \{\}} \quad (86)$$

$$\frac{\begin{array}{l} decs \vdash con' \equiv \mu_i[con] : \Omega \\ decs \vdash con : \Omega^n \Rightarrow \Omega^n \quad 1 \leq i \leq n \\ decs \vdash exp : \mathbf{Unroll}[\mu_i[con]] \end{array}}{decs \vdash \mathbf{roll}_i^{con'} \ exp : \mu_i[con]} \quad (87)$$

$$\begin{array}{c}
\text{decs} \vdash \text{con}' \equiv \mu_i [\text{con}] : \Omega \\
\text{decs} \vdash \text{con} : \Omega^n \Rightarrow \Omega^n \quad 1 \leq i \leq n \\
\text{decs} \vdash \text{exp} : \mu_i [\text{con}] \\
\hline
\text{decs} \vdash \mathbf{unroll}_i^{\text{con}'} \text{exp} : \mathbf{Unroll} [\mu_i [\text{con}]]
\end{array} \tag{88}$$

$$\begin{array}{c}
\text{decs} \vdash \text{con} \equiv \Sigma (\text{con}_1, \dots, \text{con}_n) : \Omega \\
1 \leq i \leq n \quad \text{decs} \vdash \text{exp} : \text{con}_i \\
\hline
\text{decs} \vdash \mathbf{inj}_i^{\text{con}} \text{exp} : \text{con}
\end{array} \tag{89}$$

$$\begin{array}{c}
\text{decs} \vdash \text{exp} : \Sigma (\text{con}_1, \dots, \text{con}_n) \quad 1 \leq i \leq n \\
\hline
\text{decs} \vdash \mathbf{proj}_i \text{exp} : \text{con}
\end{array} \tag{90}$$

$$\begin{array}{c}
\text{decs} \vdash \text{name} : \text{con Name} \quad \text{decs} \vdash \text{exp} : \text{con} \\
\hline
\text{decs} \vdash \mathbf{tag}^{\text{name}} \text{exp} : \mathbf{Any}
\end{array} \tag{91}$$

$$\begin{array}{c}
\text{decs} \vdash \text{name} : \text{con Name} \quad \text{decs} \vdash \text{exp} : \mathbf{Any} \\
\hline
\text{decs} \vdash \mathbf{untag}^{\text{name}} \text{exp} : \text{con}
\end{array} \tag{92}$$

$$\begin{array}{c}
\text{decs} \vdash \text{con} \equiv \Sigma (\text{con}_1, \dots, \text{con}_n) : \Omega \\
\text{decs} \vdash \text{exp} : \text{con} \\
\text{decs} \vdash \text{exp}_1 : \text{con} \rightarrow \text{con}' \quad \dots \quad \text{decs} \vdash \text{exp}_n : \text{con} \rightarrow \text{con}' \\
\hline
\text{decs} \vdash \mathbf{case}^{\text{con}} (\text{exp}_1, \dots, \text{exp}_n) \mathbf{of} \text{exp} \mathbf{end} : \text{con}'
\end{array} \tag{93}$$

Rule 93: Note that the case analysis does not do any destructuring of the sum value.

$$\begin{array}{c}
\text{decs} \vdash \text{mod} : [\text{lbl}_1 \triangleright \text{dec}_1, \dots, \text{lbl}_m \triangleright \text{dec}_m, \text{lbl} \triangleright \text{var} : \text{con}', \text{sdecs}] \\
\text{decs}, \text{dec}_1, \dots, \text{dec}_m \vdash \text{con} \equiv \text{con}' : \Omega \\
\text{decs} \vdash \text{con} : \Omega \\
\hline
\text{decs} \vdash \text{mod.lbl} : \text{con}
\end{array} \tag{94}$$

Rule 94: A projection is only well-formed if the result is typable. If *mod* is valuable, the projection is always well-formed.

$$\begin{array}{c}
\text{decs} \vdash \text{exp} : \text{con} \rightarrow \text{con}' \\
\hline
\text{decs} \vdash \text{exp} : \text{con} \rightarrow \text{con}'
\end{array} \tag{95}$$

Rule 95: Since we only allow a total arrow at the outermost level of a constructor, this rule suffices to completely describe the “subtyping” of constructors.

$$\frac{decs \vdash exp : con' \quad decs \vdash con \equiv con' : \Omega}{decs \vdash exp : con} \quad (96)$$

Rule 96: We could remove this non-syntax-directed rule by adding explicit constructor-equivalence tests to many of the the above rules.

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

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot \text{ ok}} \quad (97)$$

$$\frac{\begin{array}{l} decs \vdash dec \text{ ok} \\ decs, dec \vdash sdecs \text{ ok} \\ lbl \notin \text{dom}(sdecs) \end{array}}{decs \vdash lbl \triangleright dec, sdecs \text{ ok}} \quad (98)$$

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

$$\frac{}{decs \vdash \cdot \leq \cdot} \quad (99)$$

$$\frac{dec \vdash dec \leq dec' \quad decs, dec \vdash sdecs \leq sdecs'}{decs \vdash lbl \triangleright dec, sdecs \leq lbl \triangleright dec', sdecs'} \quad (100)$$

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

$$\frac{}{decs \vdash \cdot \equiv \cdot} \quad (101)$$

$$\frac{dec \vdash dec \equiv dec' \quad decs, dec \vdash sdecs \equiv sdecs'}{decs \vdash lbl \triangleright dec, sdecs \equiv lbl \triangleright dec', sdecs'} \quad (102)$$

$$\frac{\begin{array}{l} decs \vdash sdecs, lbl \triangleright dec, lbl' \triangleright dec', sdecs' \text{ ok} \\ \text{bound}(dec') \cap (\text{FV}(dec) \cup \text{FTV}(dec)) = \emptyset \\ \text{bound}(dec) \cap (\text{FV}(dec') \cup \text{FTV}(dec')) \end{array}}{decs \vdash sdecs, lbl \triangleright dec, lbl' \triangleright dec', sdecs' \equiv sdecs, lbl' \triangleright dec', lbl \triangleright dec, sdecs'} \quad (103)$$

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

$$\frac{}{decs \vdash \cdot : \cdot} \quad (104)$$

$$\frac{decs \vdash bnd : dec \quad decs, dec \vdash sbnds : sdecs}{decs \vdash lbl \triangleright bnd, sbnds : lbl \triangleright dec, sdecs} \quad (105)$$

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

$$\frac{decs \vdash sdecs \text{ ok}}{decs \vdash [sdecs] : \text{Sig}} \quad (106)$$

$$\frac{decs, var:sig \vdash sig' : \text{Sig}}{decs \vdash (var:sig \rightarrow sig') : \text{Sig}} \quad (107)$$

$$\frac{decs, var:sig \vdash sig' : \text{Sig}}{decs \vdash (var:sig \rightarrow sig') : \text{Sig}} \quad (108)$$

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

$$\frac{decs \vdash sdecs \leq sdecs'}{decs \vdash [sdecs] \leq [sdecs'] : \text{Sig}} \quad (109)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var:sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash (var:sig_1 \rightarrow sig'_1) \leq (var:sig_2 \rightarrow sig'_2) : \text{Sig}} \quad (110)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var:sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash (var:sig_1 \rightarrow sig'_1) \leq (var:sig_2 \rightarrow sig'_2) : \text{Sig}} \quad (111)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var:sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash (var:sig_1 \rightarrow sig'_1) \leq (var:sig_2 \rightarrow sig'_2) : \text{Sig}} \quad (112)$$

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

$$\frac{decs \vdash sdecs \equiv sdecs'}{decs \vdash [sdecs] \equiv [sdecs'] : \text{Sig}} \quad (113)$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \text{Sig} \quad decs, var: sig_1 \vdash sig'_1 \equiv sig'_2 : \text{Sig}}{decs \vdash (var: sig_1 \rightarrow sig'_1) \equiv (var: sig_2 \rightarrow sig'_2) : \text{Sig}} \quad (114)$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \text{Sig} \quad decs, var: sig_1 \vdash sig'_1 \equiv sig'_2 : \text{Sig}}{decs \vdash (var: sig_1 \rightarrow sig'_1) \equiv (var: sig_2 \rightarrow sig'_2) : \text{Sig}} \quad (115)$$

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

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var: sig, decs''}{decs \vdash var : sig} \quad (116)$$

$$\frac{decs \vdash sbnds : sdecs}{decs \vdash [sbnds] : [sdecs]} \quad (117)$$

$$\frac{decs, var: sig \vdash mod : sig'}{decs \vdash (\lambda var: sig. mod) : (var: sig \rightarrow sig')} \quad (118)$$

$$\frac{decs, var: sig \vdash mod \downarrow sig'}{decs \vdash (\lambda var: sig. mod) : (var: sig \rightarrow sig')} \quad (119)$$

$$\frac{decs \vdash mod : (sig' \rightarrow sig) \quad decs \vdash mod' : sig'}{decs \vdash mod mod' : sig} \quad (120)$$

Rule 120: 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 valuable, dependencies can always be eliminated.

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \dots, lbl_m \triangleright dec_m, lbl \triangleright var: sig', sdecs] \\ decs, dec_1, \dots, dec_m \vdash sig \equiv sig' : \text{Sig} \\ decs \vdash sig : \text{Sig} \end{array}}{decs \vdash mod.lbl : sig} \quad (121)$$

Rule 121: A projection is only well-formed if the result can be given a signature in the ambient context. If mod is valuable, the projection is always well-formed.

$$\frac{decs \vdash mod : sig}{decs \vdash mod : sig : sig} \quad (122)$$

Rule 122: Ascription of a signature to a module can permanently forget type abbreviations.

$$\frac{decs \vdash mod \downarrow [sdecs, lbl \triangleright var : kind, sdecs']}{decs \vdash mod : [sdecs, lbl \triangleright var : kind = mod.lbl, sdecs']} \quad (123)$$

Rule 123: The “self” rule. If $mod.lbls$ specifies a type and mod has a well-defined value (is valuable) then $mod.lbls \equiv mod.lbls$; we add this fact to the signature.

$$\frac{decs \vdash mod \downarrow [sdecs, lbl \triangleright var : sig, sdecs'] \quad decs \vdash mod.lbl : sig'}{decs \vdash mod : [sdecs, lbl \triangleright var : sig', sdecs']} \quad (124)$$

Rule 124: Allows the “self” rule to be applied to substructures.

$$\frac{decs \vdash mod : sig' \quad decs \vdash sig' \leq sig : \mathbf{Sig}}{decs \vdash mod : sig} \quad (125)$$

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

$$\frac{\vdash decs \mathbf{ok}}{decs \vdash \cdot \mathbf{ok}} \quad (126)$$

$$\frac{decs \vdash sig : \mathbf{Sig} \quad decs, var : sig \vdash tdecs \mathbf{ok} \quad lbl \notin \text{dom}(tdecs)}{decs \vdash lbl \triangleright var : sig, tdecs \mathbf{ok}} \quad (127)$$

$$\frac{decs \vdash sig : \mathbf{Sig} \quad decs \vdash tdecs \mathbf{ok} \quad lbl \notin \text{dom}(tdecs)}{decs \vdash lbl \triangleright var : \mathbf{Sig} = sig, tdecs \mathbf{ok}} \quad (128)$$

$$\boxed{decs \vdash tbnds : tdecs}$$

$$\frac{decs \vdash \text{ok}}{decs \vdash \cdot : \cdot} \quad (129)$$

$$\frac{decs \vdash mod : sig \quad decs, var:sig \vdash tbnds : tdecs}{decs \vdash lbl \triangleright var = mod, tbnds : lbl \triangleright var:sig, tdecs} \quad (130)$$

$$\frac{decs \vdash sig \equiv sig' : \text{Sig} \quad decs \vdash tbnds : tdecs \quad var \notin \text{bound}(decs)}{decs \vdash lbl \triangleright var = sig, tbnds : lbl \triangleright var:\text{Sig} = sig', tdecs} \quad (131)$$

5 Static Semantics: Valuability

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.

In the translation, only EL expressions whose translation is valuable will be polymorphically generalized. This means that general function applications will not be generalized, but constructor applications may be (since constructors have total arrows while general functions have partial arrows).

5.1 Judgment Forms

<i>Judgment...</i>	<i>Meaning...</i>
$decs \vdash exp \downarrow con$	exp is a valuable expression of type con
$decs \vdash mod \downarrow sig$	mod is a valuable module with signature sig
$decs \vdash exp \downarrow$	exp is a valuable expression
$decs \vdash mod \downarrow$	mod is a valuable module

5.2 Syntactic Values

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

$$\begin{aligned}
exp_v &::= scon \\
&| \{rbnds_v\} \\
&| \text{fix } fbnds \text{ in } var \text{ end} \\
&| inj_i^{con} exp_v \\
&| tag^{name} exp_v \\
rbnds_v &::= \cdot \\
&| rbnds_v, rbnd_v \\
rbnd_v &::= lbl \triangleright exp_v \\
mod_v &::= [sbnds_v] \\
&| (\lambda var : sig. mod) \\
bnd_v &::= var = exp_v \\
&| var = mod_v \\
&| var = con \\
sbnds_v &::= \cdot \\
&| sbnds_v, sbnd_v \\
sbnd_v &::= lbl \triangleright bnd_v \\
val &::= exp_v \\
&| mod_v \\
&| con
\end{aligned}$$

5.3 Inference Rules

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

$$\frac{}{decs \vdash exp_v \downarrow} \quad (132)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \quad (133)$$

$$\frac{decs \vdash exp_1 \downarrow \quad con' \rightarrow con \quad decs \vdash exp_2 \downarrow}{decs \vdash exp_1 exp_2 \downarrow} \quad (134)$$

$$\frac{decs \vdash exp_1 \downarrow \quad \cdots \quad decs \vdash exp_n \downarrow}{decs \vdash \{lbl_1 \triangleright exp_1, \dots, lbl_n \triangleright exp_n\} \downarrow} \quad (135)$$

$$\frac{decs \vdash mod \downarrow sig \quad decs, var:sig \vdash exp \downarrow}{decs \vdash \text{let } var = mod \text{ in } exp \text{ end} \downarrow} \quad (136)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \text{roll}_i^{con} exp \downarrow} \quad (137)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \text{unroll}_i^{con} exp \downarrow} \quad (138)$$

$$\frac{decs \vdash exp \downarrow \Sigma(con)}{decs \vdash \text{proj}_1 exp \downarrow} \quad (139)$$

Rule 139: Projection from a single-element sum type is total.

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

$$\overline{decs \vdash \cdot \downarrow} \quad (140)$$

$$\frac{decs \vdash exp \downarrow con \quad decs, var:consbnds \vdash \downarrow}{decs \vdash lbl \triangleright var = exp, sbnds \downarrow} \quad (141)$$

$$\frac{decs, var:knd \langle = con \rangle \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var = knd \langle = con \rangle, sbnds \downarrow} \quad (142)$$

$$\frac{decs \vdash mod \downarrow sig \quad decs, var:sig \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var = mod, sbnds \downarrow} \quad (143)$$

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

$$\overline{decs \vdash [sbnds] \downarrow} \quad (144)$$

$$\overline{decs \vdash var \downarrow} \quad (145)$$

$$\frac{decs \vdash mod \downarrow (sig' \rightarrow sig) \quad decs \vdash mod' \downarrow}{decs \vdash mod mod' \downarrow} \quad (146)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \quad (147)$$

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

$$\frac{decs \vdash exp : con \quad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \quad (148)$$

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

$$\frac{decs \vdash mod : sig \quad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \quad (149)$$

6 Dynamic Semantics

6.1 Introduction

The dynamic semantics is written as a natural semantics, but could (and probably should) easily be turned into the specification of an abstract machine.

The components of the judgments include:

- Δ , a context (*decs*) giving the types of locations and exception names. This is used to choose “new” locations and exception names, and to allow a proof of type-preservation.
- σ , a store mapping the locations in $\text{bound}(\Delta)$ to values.
- E , a “stack” of frames which all together represent a continuation or evaluation context. If F ranges over frames (see Section 6.2), then we define the grammar for E by

$$E ::= [] \\ | E \circ F$$

As a technical convenience, we let R range over a subset of frames called “raise frames.” These are the frames which propagate all exceptions.

- ans , an “answer” resulting from evaluating a core or module expression, with the following grammar:

$$\begin{array}{ll} ans ::= & \text{VALUE}(val) \quad (\text{Result is a value}) \\ & | \text{UNCAUGHT}(exp_v) \quad (\text{Uncaught exception}) \\ & | \text{FAIL} \quad (\text{Uncaught failure}) \end{array}$$

- Ans , an “answer” resulting from evaluating an entire program (*tbnds*), with the following grammar:

$$\begin{array}{ll} Ans ::= & \text{VALUES}(tbnds) \quad (\text{Program result}) \\ & | \text{UNCAUGHT}(exp_v) \quad (\text{Uncaught exception}) \\ & | \text{FAIL} \quad (\text{Uncaught failure}) \end{array}$$

6.2 Frames

$$\begin{aligned}
 F ::= & \\
 & \mid \text{[] } exp \\
 & \mid exp_v \text{ []} \\
 & \mid \{rbnds_v, lbl \triangleright \text{[]}, rbnds\} \\
 & \mid \text{[]} \# lbl \\
 & \mid \text{handle [] with } exp \\
 & \mid \text{raise []} \\
 & \mid \text{catch [] with } exp \\
 & \mid \text{let } var = \text{[] in } exp \text{ end} \\
 & \mid \text{ref}^{con} \text{ []} \\
 & \mid \text{get []} \\
 & \mid \text{set []} \\
 & \mid \text{roll}_i^{con} \text{ []} \\
 & \mid \text{unroll}_i^{con} \text{ []} \\
 & \mid \text{inj}_i^{con} \text{ []} \\
 & \mid \text{proj}_i \text{ []} \\
 & \mid \text{tag}^{name} \text{ []} \\
 & \mid \text{untag}^{name} \text{ []} \\
 & \mid \text{case}^{con} (exp_1, \dots, exp_n) \text{ of [] end} \\
 & \mid [sbnds_v, lbl \triangleright var = \text{[]}, sbnds] \\
 & \mid \text{[] } mod \\
 & \mid mod_v \text{ []} \\
 & \mid \text{[]}.lbl \\
 & \mid \text{[]}:sig
 \end{aligned}$$

6.3 Raise Frames

$$\begin{array}{lcl}
 R ::= & [] & \\
 & | \quad [] \text{ exp} & \\
 & | \quad \text{exp}_v \quad [] & \\
 & | \quad \{rbnds_v, lbl \triangleright [], rbnds\} & \\
 & | \quad [] \# lbl & \\
 & | \quad \text{raise } [] & \\
 & | \quad \text{let } var = [] \text{ in exp end} & \\
 & | \quad \text{ref}^{con} \quad [] & \\
 & | \quad \text{get } [] & \\
 & | \quad \text{set } [] & \\
 & | \quad \text{roll}_i^{con} \quad [] & \\
 & | \quad \text{unroll}_i^{con} \quad [] & \\
 & | \quad \text{inj}_i^{con} \quad [] & \\
 & | \quad \text{proj}_i \quad [] & \\
 & | \quad \text{tag}^{name} \quad [] & \\
 & | \quad \text{untag}^{name} \quad [] & \\
 & | \quad \text{case}^{con} (exp_1, \dots, exp_n) \text{ of } [] \text{ end} & \\
 & | \quad [sbnds_v, lbl \triangleright var = [], sbnds] & \\
 & | \quad [] \text{ mod} & \\
 & | \quad \text{mod}_v \quad [] & \\
 & | \quad [] . lbl & \\
 & | \quad [] : sig &
 \end{array}$$

6.4 Judgment Forms

<i>Judgment...</i>	<i>Meaning...</i>
$\Delta, \sigma, E \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}$	Evaluation of expression
$\Delta, \sigma, E \vdash \text{mod} \Downarrow \Delta', \sigma', \text{ans}$	Evaluation of module expression
$\text{val} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', \text{ans}$	Plug value back into most recent frame
$\text{raise } \text{exp}_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', \text{ans}$	Propagate exception
$\text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', \text{ans}$	Propagate failure
$\Delta, \sigma \vdash \text{tbnds} \Downarrow \Delta', \sigma', \text{Ans}$	Evaluation of entire program

6.5 Inference Rules

$$\boxed{\Delta, \sigma, E \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}$$

$$\frac{\text{val} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{val} \Downarrow \Delta', \sigma', \text{ans}} \quad (150)$$

$$\frac{\Delta, \sigma, E \circ [\langle \rangle \text{exp}'] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{exp exp}' \Downarrow \Delta', \sigma', \text{ans}} \quad (151)$$

$$\frac{\Delta, \sigma, E \circ [\{\text{lbl} \triangleright \langle \rangle, \text{rbnds}\}] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \{\text{lbl} \triangleright \text{exp}, \text{rbnds}\} \Downarrow \Delta', \sigma', \text{ans}} \quad (152)$$

Rule 152: A record value can evaluate to itself in one step (via Rule 150) or here component-by-component.

$$\frac{\Delta, \sigma, E \circ [\langle \rangle \# \text{lbl}] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{exp} \# \text{lbl} \Downarrow \Delta', \sigma', \text{ans}} \quad (153)$$

$$\frac{\Delta, \sigma, E \circ [\text{handle } \langle \rangle \text{ with exp}'] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{handle exp with exp}' \Downarrow \Delta', \sigma', \text{ans}} \quad (154)$$

$$\frac{\Delta, \sigma, E \circ [\text{raise } \langle \rangle] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{raise exp} \Downarrow \Delta', \sigma', \text{ans}} \quad (155)$$

$$\frac{\Delta, \sigma, E \circ [\text{catch } \langle \rangle \text{ with exp}'] \vdash \text{exp} \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{catch exp with exp}' \Downarrow \Delta', \sigma', \text{ans}} \quad (156)$$

$$\frac{\text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', \text{ans}}{\Delta, \sigma, E \vdash \text{fail} \Downarrow \Delta', \sigma', \text{ans}} \quad (157)$$

$$\frac{\Delta, \sigma, E \circ [\text{let } var = [] \text{ in } exp \text{ end}] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{let } var = mod \text{ in } exp \text{ end} \Downarrow \Delta', \sigma', ans} \quad (158)$$

$$\frac{\begin{array}{l} name \notin \text{dom}(\Delta) \\ \{\text{mk} \triangleright \lambda var : con.\text{tag}^{name} var, \\ \text{km} \triangleright \lambda var : \text{Any}.\text{untag}^{name} var\} \vdash \Delta[name : con \text{ Name}], \sigma, E \Downarrow \Delta', \sigma', ans \end{array}}{\Delta, \sigma, E \vdash \text{new_stamp}[con] \Downarrow \Delta', \sigma', ans} \quad (159)$$

$$\frac{\Delta, \sigma, E \circ [\text{ref}^{con} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{ref}^{con} exp \Downarrow \Delta', \sigma', ans} \quad (160)$$

$$\frac{\Delta, \sigma, E \circ [\text{get} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{get } exp \Downarrow \Delta', \sigma', ans} \quad (161)$$

$$\frac{\Delta, \sigma, E \circ [\text{set} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{set } exp \Downarrow \Delta', \sigma', ans} \quad (162)$$

$$\frac{\Delta, \sigma, E \circ [\text{roll}_i^{con} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{roll}_i^{con} exp \Downarrow \Delta', \sigma', ans} \quad (163)$$

$$\frac{\Delta, \sigma, E \circ [\text{unroll}_i^{con} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{unroll}_i^{con} exp \Downarrow \Delta', \sigma', ans} \quad (164)$$

$$\frac{\Delta, \sigma, E \circ [\text{inj}_i^{con} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{inj}_i^{con} exp \Downarrow \Delta', \sigma', ans} \quad (165)$$

$$\frac{\Delta, \sigma, E \circ [\text{proj}_i []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{proj}_i exp \Downarrow \Delta', \sigma', ans} \quad (166)$$

$$\frac{\Delta, \sigma, E \circ [\text{inj}_{[]}^{name}] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{tag}^{name} exp \Downarrow \Delta', \sigma', ans} \quad (167)$$

$$\frac{\Delta, \sigma, E \circ [\text{untag}^{name} []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{untag}^{name} exp \Downarrow \Delta', \sigma', ans} \quad (168)$$

$$\frac{\Delta, \sigma, E \circ [\text{case}^{con}(exp_1, \dots, exp_n) \text{ of } [] \text{ end}] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \text{case}^{con}(exp_1, \dots, exp_n) \text{ of } exp \text{ end} \Downarrow \Delta', \sigma', ans} \quad (169)$$

$$\frac{\Delta, \sigma, E \circ [[].\text{lbl}] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod.\text{lbl} \Downarrow \Delta', \sigma', ans} \quad (170)$$

$$\boxed{\Delta, \sigma, E \vdash mod \Downarrow \Delta', \sigma', ans}$$

$$\frac{\Delta, \sigma, E \circ [[lbl \triangleright var = [], sbnds]] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var = exp, sbnds] \Downarrow \Delta', \sigma', ans} \quad (171)$$

$$\frac{con \vdash \Delta, \sigma, E \circ [[lbl \triangleright var = [], sbnds]] \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var = con, sbnds] \Downarrow \Delta', \sigma', ans} \quad (172)$$

$$\frac{\Delta, \sigma, E \circ [[lbl \triangleright var = [], sbnds]] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var = mod, sbnds] \Downarrow \Delta', \sigma', ans} \quad (173)$$

$$\frac{\Delta, \sigma, E \circ [[] mod'] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod mod' \Downarrow \Delta', \sigma', ans} \quad (174)$$

$$\frac{\Delta, \sigma, E \circ [[] . lbl] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod . lbl \Downarrow \Delta', \sigma', ans} \quad (175)$$

$$\frac{\Delta, \sigma, E \circ [[] : sig] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod : lbl \Downarrow \Delta', \sigma', ans} \quad (176)$$

$$\boxed{val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\overline{val \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \text{VALUE}(val)} \quad (177)$$

$$\frac{\Delta, \sigma, E \circ [exp_v []] \vdash exp \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [[] exp] \Downarrow \Delta', \sigma', ans} \quad (178)$$

$$\frac{\begin{array}{c} exp_v^k = \text{fix } (lbl_i = (var'_i; var_i) \mapsto exp_i)_{i=1}^n \text{ in } var_k \text{ end} \quad (\forall k \in 1..n) \\ 1 \leq i \leq n \\ \Delta, \sigma, E \vdash [exp_v^1 / var'_1] \cdots [exp_v^n / var'_n] [exp_v / var_i] exp_i \Downarrow \Delta', \sigma', ans \end{array}}{exp_v \vdash \Delta, \sigma, E \circ [exp_v^i []] \Downarrow \Delta', \sigma', ans} \quad (179)$$

$$\frac{\Delta, \sigma, E \circ [\{rbnds_v, lbl \triangleright exp_v, lbl' \triangleright [], rbnds\}] \vdash exp' \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\{rbnds_v, lbl \triangleright [], lbl' \triangleright exp', rbnds\}] \Downarrow \Delta', \sigma', ans} \quad (180)$$

$$\frac{\{rbnds_v, lbl \triangleright exp_v\} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\{rbnds_v, lbl \triangleright []\}] \Downarrow \Delta', \sigma', ans} \quad (181)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\{rbnds_v, lbl \triangleright exp_v, rbnds_v'\} \vdash \Delta, \sigma, E \circ [\#lbl] \Downarrow \Delta', \sigma', ans} \quad (182)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{handle } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (183)$$

$$\frac{\text{raise } exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{raise } []] \Downarrow \Delta', \sigma', ans} \quad (184)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{catch } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (185)$$

$$\frac{loc \notin \text{dom}(\Delta) \quad loc \vdash \Delta[loc:con], \sigma[loc \mapsto exp_v], E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{ref}^{con} []] \Downarrow \Delta', \sigma', ans} \quad (186)$$

$$\frac{exp_v = \sigma(loc) \quad exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{get } con []] \Downarrow \Delta', \sigma', ans} \quad (187)$$

$$\frac{\{\} \vdash \Delta, \sigma[loc \mapsto exp_v], E \Downarrow \Delta', \sigma', ans}{\{1 \triangleright loc, 2 \triangleright exp_v\} \vdash \Delta, \sigma, E \circ [\text{set } con []] \Downarrow \Delta', \sigma', ans} \quad (188)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{roll}_i^{con} []] \Downarrow \Delta', \sigma', ans} \quad (189)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{unroll}_i^{con} []] \Downarrow \Delta', \sigma', ans} \quad (190)$$

$$\frac{\text{inj}_i^{con} exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{inj}_i^{con} []] \Downarrow \Delta', \sigma', ans} \quad (191)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{inj}_i^{con} exp_v \vdash \Delta, \sigma, E \circ [\text{proj}_{con'} i []] \Downarrow \Delta', \sigma', ans} \quad (192)$$

$$\frac{i \neq j \quad \text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{inj}_i^{con} exp_v \vdash \Delta, \sigma, E \circ [\text{proj}_{con'} j []] \Downarrow \Delta', \sigma', ans} \quad (193)$$

$$\frac{\text{tag}^{name} exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_v \vdash \Delta, \sigma, E \circ [\text{tag}^{name} []] \Downarrow \Delta', \sigma', ans} \quad (194)$$

$$\frac{exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{tag^{name} exp_v \vdash \Delta, \sigma, E \circ [untag^{name} []] \Downarrow \Delta', \sigma', ans} \quad (195)$$

$$\frac{name \neq name' \quad fail \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{tag^{name} exp_v \vdash \Delta, \sigma, E \circ [untag^{name'} []] \Downarrow \Delta', \sigma', ans} \quad (196)$$

$$\frac{\Delta, \sigma, E \vdash exp_i exp_v \Downarrow \Delta', \sigma', ans}{inj_i^{con} exp_v \vdash \Delta, \sigma, E \circ [case^{con'}(exp_1, \dots, exp_n) \text{ of } [] \text{ end}] \Downarrow \Delta', \sigma', ans} \quad (197)$$

$$\frac{\Delta, \sigma, E \vdash [mod_v/var]exp \Downarrow \Delta', \sigma', ans}{mod_v \vdash \Delta, \sigma, E \circ [\text{let } var = [] \text{ in } exp \text{ end}] \Downarrow \Delta', \sigma', ans} \quad (198)$$

$$\frac{\Delta, \sigma, E \circ [mod_v []] \vdash mod \Downarrow \Delta', \sigma', ans}{mod_v \vdash \Delta, \sigma, E \circ [[] mod'] \Downarrow \Delta', \sigma', ans} \quad (199)$$

$$\frac{\Delta, \sigma, E \vdash [mod_v/var]mod \Downarrow \Delta', \sigma', ans}{mod_v \vdash \Delta, \sigma, E \circ [(\lambda var: sig. mod) []] \Downarrow \Delta', \sigma', ans} \quad (200)$$

$$\frac{val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{[sbnds_v, lbl \triangleright val, sbnds_v'] \vdash \Delta, \sigma, E \circ [[] . lbl] \Downarrow \Delta', \sigma', ans} \quad (201)$$

$$\frac{mod_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{mod_v \vdash \Delta, \sigma, E \circ [[] : sig] \Downarrow \Delta', \sigma', ans} \quad (202)$$

Rule 202: Forgetting of type abbreviations has no run-time effect.

$$\frac{\Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = val, lbl' \triangleright var' = [], [val/var]sbnds]] \vdash [val/var]exp \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = [], lbl' \triangleright var' = exp, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \quad (203)$$

$$\frac{\Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = val, lbl' \triangleright var' = [], [val/var]sbnds]] \vdash [val/var]mod \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = [], lbl' \triangleright var' = mod, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \quad (204)$$

$$\frac{con \vdash \Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = val, lbl' \triangleright var' = [], [val/var]sbnds]] \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = [], lbl' \triangleright var' = con, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \quad (205)$$

$$\frac{[sbnds_v, lbl \triangleright var = val] \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_v, lbl \triangleright var = []]] \Downarrow \Delta', \sigma', ans} \quad (206)$$

$$\boxed{\text{raise } exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\frac{}{\text{raise } exp_v \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \text{UNCAUGHT}(exp_v)} \quad (207)$$

$$\frac{\text{raise } exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{raise } exp_v \vdash \Delta, \sigma, E \circ R \Downarrow \Delta', \sigma', ans} \quad (208)$$

$$\frac{\Delta, \sigma, E \vdash exp \ exp_v \Downarrow \Delta', \sigma', ans}{\text{raise } exp_v \vdash \Delta, \sigma, E \circ [\text{handle } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (209)$$

$$\frac{\text{raise } exp_v \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{raise } exp_v \vdash \Delta, \sigma, E \circ [\text{catch } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (210)$$

$$\boxed{\text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\frac{}{\text{fail} \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \text{FAIL}} \quad (211)$$

$$\frac{\text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{fail} \vdash \Delta, \sigma, E \circ R \Downarrow \Delta', \sigma', ans} \quad (212)$$

$$\frac{\text{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\text{fail} \vdash \Delta, \sigma, E \circ [\text{handle } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (213)$$

$$\frac{\Delta, \sigma, E \vdash exp \Downarrow \Delta', \sigma', ans}{\text{fail} \vdash \Delta, \sigma, E \circ [\text{catch } [] \text{ with } exp] \Downarrow \Delta', \sigma', ans} \quad (214)$$

$$\boxed{\Delta, \sigma \vdash tbnds \Downarrow \Delta', \sigma', Ans}$$

$$\frac{}{\Delta, \sigma \vdash \cdot \Downarrow \Delta, \sigma, \text{VALUES}(\cdot)} \quad (215)$$

$$\frac{\begin{array}{l} \Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \text{VALUE}(mod_v) \\ \Delta', \sigma' \vdash [mod_v/var]tbnds \Downarrow \Delta'', \sigma'', \text{VALUES}(tbnds') \end{array}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{VALUES}(lbl \triangleright var = mod_v, tbnds')} \quad (216)$$

$$\frac{\begin{array}{l} \Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \text{VALUE}(mod_v) \\ \Delta', \sigma' \vdash [mod_v/var]tbnds \Downarrow \Delta'', \sigma'', \text{UNCAUGHT}(exp_v) \end{array}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{UNCAUGHT}(exp_v)} \quad (217)$$

$$\frac{\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \text{VALUE}(mod_v)}{\Delta', \sigma' \vdash [mod_v/var]tbnds \Downarrow \Delta'', \sigma'', \text{FAIL}}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{FAIL}} \quad (218)$$

$$\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \text{UNCAUGHT}(exp_v)}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{UNCAUGHT}(exp_v)} \quad (219)$$

$$\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \text{FAIL}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{FAIL}} \quad (220)$$

$$\frac{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \text{VALUES}(tbnds')}{\Delta, \sigma \vdash tbnds \Downarrow \Delta'', \sigma'', \text{VALUES}(lbl \triangleright var = sig, tbnds')} \quad (221)$$

7 External Language

7.1 Notation

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

7.2 Grammar of the Abstract Syntax

```
expr ::= scon
        | longid
        | {lab1 = expr1, ..., labn = exprn}
        | let strdec in expr end
        | expr expr'
        | expr : ty
        | expr handle match
        | raise expr
        | fn match

mrule ::= pat => expr

match ::= mrule
        | mrule | match

strdec ::= val (tyvar1, ..., tyvarn) pat = exp
        | val (tyvar1, ..., tyvarn) rec pat = exp
        | strdec1 strdec2
        | open longid1 ... longidn
        | exception id
        | exception id of ty
        | exception id = longid
        | local strdec1 in strdec2 end
        | type tybind
        | datatype datbind
        | structure strbind
        | functor funbind
```

```

tybind ::= ⟨⟨tyvar1, …, tyvarn⟩⟩ tycon = ty ⟨⟨and tybind⟩⟩
datbind ::= ⟨⟨tyvar1, …, tyvarn⟩⟩ tycon = conbind ⟨and datbind⟩
conbind ::= id ⟨of ty⟩ ⟨⟨l conbind⟩⟩

strex ::= struct strdec end
          | funid (strex)
          | strex : sigexp
          | strex :> sigexp
          | let strdec in strex end

spec ::= val id : ty
          | type typdesc
          | datatype datbind
          | exception id
          | exception id of ty
          | structure strid : sigexp
          | functor funid (strid : sigexp) : sigexp'
          | include sigexp
          | spec1 spec2
          | spec sharing type longid1 = longid2

typdesc ::= ⟨⟨tyvar1, …, tyvarn⟩⟩ tycon ⟨⟨and typdesc⟩⟩
          | ⟨⟨tyvar1, …, tyvarn⟩⟩ tycon = ty ⟨⟨and typdesc⟩⟩

sigexp ::= sig spec end
          | sigid
          | sigexp where type ⟨⟨tyvar1, …, tyvarn⟩⟩ longtycon = ty
          | sigexp where type ⟨⟨tyvar1, …, tyvarn⟩⟩ longtycon = own ty

pat ::= scon
        | longid
        | -
        | pat : ty
        | longid pat
        | {lab1 = pat1, …, labn = patn⟨, …⟩}
        | pat1 as pat2
        | ref pat

```

$$\begin{aligned}
ty &::= tyvar \\
&| \{lab_1 : expr_1, \dots, lab_n : expr_n\} \\
&| \langle (ty_1, \dots, ty_n) \rangle longtycon \\
&| ty \rightarrow ty' \\
sigbind &::= sigid = sigexp \langle \mathbf{and} \ sigbind \rangle \\
strbind &::= strid = strexp \langle \mathbf{and} \ strbind \rangle \\
funbind &::= funid (strid : sigexp) = strexp \langle \mathbf{and} \ funbind \rangle \\
topdec &::= \mathbf{signature} \ sigbind \\
&| \mathbf{structure} \ strbind \\
&| \mathbf{functor} \ funbind \\
&| topdec_1 \ topdec_2
\end{aligned}$$

7.3 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 occurring 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

$$\{lab_1=id_1, \dots, lab_n=id_n\}$$

and the expression must be of the form

$$\{lab_1=fn\ match_1, \dots, lab_n=fn\ match_n\}.$$

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 scope the definition of that datatype. This disallows such code as

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

8 The Translation

8.1 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 is encoded as a use of the modules system. Polymorphic values are translated into functors, which are explicitly instantiated with structures of types when necessary. More precisely, the functor takes a structure containing type constructors of kind Ω , and returns a structure whose single component (with label “it”) is the polymorphic value made monomorphic by instantiating it with the given types.

3. Patterns are expanded into uses of the appropriate record projections and datatype deconstructors. Thus the translation specifies a reference pattern compiler.
4. Each series of external language bindings (*strdec*) translates into a structure, containing a component for every variable bound in the external language. External language *identifiers* correspond to internal language *labels*.
5. 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.)
6. 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**.
7. 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 a structure apparent from its signature.

8.2 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 special labels “mk”, “km”, “case”, and “it” used by translation are not in the range of the overbar mapping, and 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.

- The translation maintains a translation context Γ . 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.

8.3 Initial Basis

The translation assumes the presence of a structure $basis: sig_{basis}$ serving as the initial basis for the internal language. We use the components:

$basis.\overline{bind}$	Bind exception
$basis.\overline{match}$	Match exception
$basis.\overline{bool}$	Bool datatype, with constructors $basis.\overline{true}$ and $basis.\overline{false}$
$basis.equal_{con}$	Equality on base types con for pattern-matching

8.4 Judgment Forms

<i>Judgment...</i>	<i>Meaning...</i>
$\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}$	Translation of an expression
$\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}$	Translation of a pattern match
$\Gamma \vdash \text{strdec} \rightsquigarrow \text{mod} : \text{sig}$	Translation of a declaration
$\Gamma \vdash \text{strexpr} \rightsquigarrow \text{mod} : \text{sig}$	Translation of a structure expression
$\Gamma \vdash \text{spec} \rightsquigarrow [\text{sdec}s] : \text{Sig}$	Translation of a signature specification
$\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}$	Translation of a signature expression
$\Gamma \vdash \text{topdec} \rightsquigarrow \text{tbnds} : \text{tdec}s$	Translation of a program
$\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega$	Translation of a type
$\Gamma \vdash \text{tybind} \rightsquigarrow \text{mod} : \text{sig}$	Translation of a type definition
$\Gamma \vdash \text{datbind} \rightsquigarrow \text{mod} : \text{sig}$	Translation of a datatype definition
$\Gamma \vdash \text{pat} \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig}$	Pattern Compilation
$\Gamma \vdash_{\text{ctx}} \text{lbl} \rightsquigarrow \text{path} : \Theta / \text{sig}$	Lookup in a translation context
$\Gamma, \text{path} : \text{sdec}s \vdash_{\text{sig}} \text{lbl} \rightsquigarrow \text{lbls} : \Theta$	Lookup in a signature
$\text{dec}s \vdash_{\text{sub}} \text{sig}_0 \preceq \text{sig} \rightsquigarrow$ $(\lambda \text{var}_0 : \text{sig}_0 . \text{mod}) :$ $((\text{var}_0 : \text{sig}_0) \rightarrow \text{sig}')$	Witness to EL-signature subtyping
$\text{sig} \vdash_{\text{wt}} \text{lbls} := \text{con} : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$	Signature Patching (where type)
$\text{sig} \vdash_{\text{sh}} \text{lbls} := \text{lbls}' : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$	Signature Patching (sharing type)

8.5 Inference Rules

Expressions

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

$$\frac{}{\Gamma \vdash \text{scon} \rightsquigarrow \text{scon} : \text{type}(\text{scon})} \quad (222)$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{con}}{\Gamma \vdash \text{longid} \rightsquigarrow \text{path} : \text{con}} \quad (223)$$

Rule 223: This rule handles monomorphic identifiers.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{sig} \\
\Gamma \vdash \text{sig} \leq (\text{sig}_{\text{poly}} \rightarrow [\text{it} \triangleright \text{con}]) : \text{Sig} \\
\Gamma \vdash \text{mod}_{\text{poly}} \downarrow \text{sig}_{\text{poly}} \\
\hline
\Gamma \vdash \text{longid} \rightsquigarrow \text{path}(\text{mod}_{\text{poly}}).\text{it} : \text{con}
\end{array} \tag{224}$$

Rule 224:

- 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} in Rule 224 is the structure of types that we “guess” to instantiate the polymorphic function.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{sig} \quad \Gamma \vdash \text{path.mk} : \text{con}}{\Gamma \vdash \text{longid} \rightsquigarrow \text{path} : \text{con}} \tag{225}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{sig} \quad \Gamma \vdash \text{path.mk} : \text{sig}_{\text{mk}} \\ \Gamma \vdash \text{sig}_{\text{mk}} \leq (\text{sig}_{\text{poly}} \rightarrow [\text{it} \triangleright \text{con}]) : \text{Sig} \\ \Gamma \vdash \text{mod}_{\text{poly}} \downarrow \text{sig}_{\text{poly}} \end{array}}{\Gamma \vdash \text{longid} \rightsquigarrow ((\text{path.mk})\text{mod}_{\text{poly}}).\text{it} : \text{con}} \tag{226}$$

Rules 225 and 226: 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, making the type partial in the process.

$$\frac{\Gamma \vdash \text{expr}_1 \rightsquigarrow \text{exp}_1 : \text{con}_1 \quad \dots \quad \Gamma \vdash \text{expr}_n \rightsquigarrow \text{exp}_n : \text{con}_n}{\Gamma \vdash \{\overline{\text{lab}_1} = \text{expr}_1, \dots, \overline{\text{lab}_n} = \text{expr}_n\} \rightsquigarrow \{\overline{\text{lab}_1} \triangleright \text{exp}_1, \dots, \overline{\text{lab}_n} \triangleright \text{exp}_n\} : \{\overline{\text{lab}_1} \triangleright \text{con}_1, \dots, \overline{\text{lab}_n} \triangleright \text{con}_n\}} \tag{227}$$

$$\frac{\begin{array}{c} \Gamma \vdash \text{dec} \rightsquigarrow \text{mod} : \text{sig} \\ \text{var} \notin \text{bound}(\Gamma) \quad \Gamma, \text{lbl}^* \triangleright \text{var} : \text{sig} \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}' \\ \Gamma, \text{var} : \text{sig} \vdash \text{con}' \equiv \text{con} : \Omega \quad \Gamma \vdash \text{con} : \Omega \end{array}}{\Gamma \vdash \text{let } \text{dec} \text{ in } \text{expr} \text{ end} \rightsquigarrow \text{let } \text{var} = \text{mod} \text{ in } \text{exp} \text{ end} : \text{con}} \tag{228}$$

Rule 228:

- Declarations are uniformly translated to components of a structure; the “starred structure” convention is used here to make these components accessible by name.
- This rule prohibits the type of the body from depending on abstract types defined locally—in particular, datatypes cannot escape their scope.

$$\frac{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}'' \quad \Gamma \vdash \text{expr}' \rightsquigarrow \text{exp}' : \text{con}' \quad \Gamma \vdash \text{exp} : \text{con}' \rightarrow \text{con}}{\Gamma \vdash \text{expr} \text{ expr}' \rightsquigarrow \text{exp} \text{ exp}' : \text{con}} \quad (229)$$

Rule 229: We check that the application is well-typed in the IL. (This rule works whether *exp* is partial or total.)

$$\frac{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \quad \Gamma \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \quad \Gamma \vdash \text{con} \equiv \text{con}' : \Omega}{\Gamma \vdash \text{expr} : \text{ty} \rightsquigarrow \text{exp} : \text{con}} \quad (230)$$

$$\frac{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \quad \Gamma \vdash \text{match} \rightsquigarrow \text{exp}' : \text{Any} \rightarrow \text{con} \quad \text{var} \notin \text{bound}(\Gamma)}{\Gamma \vdash \text{expr} \text{ handle } \text{match} \rightsquigarrow \text{handle } \text{exp} \text{ with } \lambda \text{ var} : \text{Any}. (\text{catch } \text{exp}' \text{ var with raise } \text{var}) : \text{con}} \quad (231)$$

Rule 231: The handling expression *exp' var* may fail if the handler pattern does not match the exception raised, in which case we propagate the exception.

$$\frac{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{Any}}{\Gamma \vdash \text{raise } \text{expr} \rightsquigarrow \text{raise } \text{exp} : \text{con}} \quad (232)$$

$$\frac{\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}_1 \rightarrow \text{con}_2 \quad \text{var} \notin \text{bound}(\Gamma)}{\Gamma \vdash \text{fn } \text{match} \rightsquigarrow \lambda \text{ var} : \text{con}_1. (\text{catch } \text{exp} \text{ var with raise } \text{basis}.\overline{\text{Match}}) : \text{con}_1 \rightarrow \text{con}_2} \quad (233)$$

Rule 233: The expression *exp var* will fail if the match fails; here we turn failure into a *basis.Match* exception.

Matches

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

$$\frac{\begin{array}{c} var, var' \notin \text{bound}(\Gamma) \\ \Gamma \vdash con' : \Omega \\ \Gamma \vdash pat \Leftarrow var' : con' \mid \text{fail} \rightsquigarrow mod : sig \\ \Gamma, lbl^* \triangleright var : sig \vdash expr \rightsquigarrow exp : con \end{array}}{\Gamma \vdash pat \Rightarrow expr \rightsquigarrow \lambda var' : con'. \text{let } var = mod \text{ in } exp \text{ end} : con' \rightarrow con} \quad (234)$$

Rule 234:

- The result of translating a match is a function that may fail if the match fails.
- We know that $\Gamma \vdash con' \rightarrow con : \Omega$ because the pattern-compilation rules create a structure with no type components.

$$\frac{\begin{array}{c} var \notin \text{bound}(\Gamma) \\ \Gamma \vdash mrule \rightsquigarrow exp : con' \rightarrow con \\ \Gamma \vdash match \rightsquigarrow exp' : con' \rightarrow con \end{array}}{\Gamma \vdash mrule \mid match \rightsquigarrow \lambda var : con'. \text{catch } exp \text{ var with } exp' \text{ var} : con' \rightarrow con} \quad (235)$$

Declarations

$$\boxed{\Gamma \vdash strdec \rightsquigarrow mod : sig}$$

$$\frac{\begin{array}{c} \Gamma \vdash expr \rightsquigarrow exp : con \\ \Gamma \vdash pat \Leftarrow exp : con \mid \text{raise basis.} \overline{\text{Bind}} \rightsquigarrow mod : sig \end{array}}{\Gamma \vdash \text{val } () \text{ pat} = expr \rightsquigarrow mod : sig} \quad (236)$$

Rule 236: This is the monomorphic, non-recursive case.

$$\begin{array}{c}
\text{lbl}^* \notin \text{dom}(\Gamma) \quad \text{lbl}' \text{ fresh} \\
\text{sig}_{poly} = [\overline{\text{tyvar}_1 \triangleright \Omega}, \dots, \overline{\text{tyvar}_n \triangleright \Omega}, \text{lbl}'_1 \triangleright \Omega, \dots, \text{lbl}'_m \triangleright \Omega] \\
\Gamma, \text{lbl}^* \triangleright \text{var}_{poly} : \text{sig}_{poly} \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \\
\Gamma, \text{lbl}^* \triangleright \text{var}_{poly} : \text{sig}_{poly} \vdash \text{pat} \Leftarrow \text{exp} : \text{con} \mid \text{raise basis.} \overline{\text{Bind}} \rightsquigarrow \text{mod} : \text{sig} \\
\Gamma, \text{lbl}^* \triangleright \text{var}_{poly} : \text{sig}_{poly} \vdash \text{mod} \Downarrow \text{sig} \\
\text{sig} = [\text{lbl}_1 \triangleright \text{con}_1, \dots, \text{lbl}_k \triangleright \text{con}_k] \\
\hline
\Gamma \vdash \mathbf{val} \ (\text{tyvar}_1, \dots, \text{tyvar}_n) \ \text{pat} = \text{expr} \rightsquigarrow \\
[\text{lbl}' \triangleright \text{var}' = (\lambda \text{var} : \text{sig}_{poly}. \text{mod}), \\
\text{lbl}_1 \triangleright (\lambda \text{var} : \text{sig}_{poly}. [\text{it} \triangleright (\text{var}'(\text{var})). \text{lbl}'_1]), \\
\vdots \\
\text{lbl}_k \triangleright (\lambda \text{var} : \text{sig}_{poly}. [\text{it} \triangleright (\text{var}'(\text{var})). \text{lbl}'_k])] : \\
[\text{lbl}' \triangleright \text{var}' : ((\text{var} : \text{sig}_{poly}) \rightarrow \text{sig}), \\
\text{lbl}_1 \triangleright ((\text{var} : \text{sig}_{poly}) \rightarrow [\text{it} \triangleright \text{con}_1]), \\
\vdots \\
\text{lbl}_k \triangleright ((\text{var} : \text{sig}_{poly}) \rightarrow [\text{it} \triangleright \text{con}_k])]
\end{array} \tag{237}$$

Rule 237:

- 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.

$$\begin{array}{c}
\text{var}_{poly} \notin \text{bound}(\Gamma) \\
\text{sig}_{poly} = [\overline{\text{tyvar}_1 \triangleright \Omega}, \dots, \overline{\text{tyvar}_n \triangleright \Omega}, \text{lbl}_1 \triangleright \Omega, \dots, \text{lbl}_m \triangleright \Omega] \\
\Gamma' = \Gamma, \text{lbl}_1^* \triangleright \text{var}_{poly} : \text{sig}_{poly}, \overline{id_1 \triangleright \text{var}'_1 : \text{con}_1 \rightarrow \text{con}'_1}, \dots, \overline{id_n \triangleright \text{var}'_n : \text{con}_n \rightarrow \text{con}'_n} \\
\Gamma' \vdash \text{match}_1 \rightsquigarrow \lambda \text{var}_1 : \text{con}_1. \text{exp}_1 : \text{con}_1 \rightarrow \text{con}'_1 \\
\vdots \\
\Gamma' \vdash \text{match}_m \rightsquigarrow \lambda \text{var}_n : \text{con}_n. \text{exp}_n : \text{con}_n \rightarrow \text{con}'_n \\
\text{exp}_k = \text{fix } \text{var}'_1 = (\text{var}_1 : \text{con}_1) \mapsto \text{exp}_1, \dots, \text{var}'_m = (\text{var}_m : \text{con}_m) \mapsto \text{exp}_m \text{ in } \text{var}'_k \text{ end} \\
\hline
\Gamma \vdash \mathbf{val} (\text{tyvar}_1, \dots, \text{tyvar}_n) \mathbf{rec} \{ (lab_i = id_i)_{i=1}^n \} = \{ (lab_i = \mathbf{fn} \text{ match}_i)_{i=1}^n \} \rightsquigarrow \\
[\overline{id_1 \triangleright (\lambda \text{var}_{poly} : \text{sig}_{poly}. [\text{it} \triangleright \text{exp}_1])}] \\
\vdots \\
[\overline{id_m \triangleright (\lambda \text{var}_{poly} : \text{sig}_{poly}. [\text{it} \triangleright \text{exp}_m])}] : \\
[\overline{id_1 \triangleright ((\text{var}_{poly} : \text{sig}_{poly}) \rightarrow [\text{it} \triangleright \text{con}_1 \rightarrow \text{con}'_1])}], \\
\vdots \\
[\overline{id_m \triangleright ((\text{var}_{poly} : \text{sig}_{poly}) \rightarrow [\text{it} \triangleright \text{con}_m \rightarrow \text{con}'_m])}]
\end{array} \tag{238}$$

Rule 238: This rule handles recursive **val** bindings. As in Rule 237, we assume that the implicit scoping of explicit type variables has been made explicit by a prepass over the EL.

$$\begin{array}{c}
\Gamma \vdash \text{strdec}_1 \rightsquigarrow \text{mod}_1 : \text{sig}_1 \\
\text{var}_1 \notin \text{bound}(\Gamma) \\
\Gamma, \text{lbl}_1^* \triangleright \text{var}_1 : \text{sig}_1 \vdash \text{strdec}_2 \rightsquigarrow \text{mod}_2 : \text{sig}_2 \\
\hline
\Gamma \vdash \text{strdec}_1 \text{ strdec}_2 \rightsquigarrow \\
[\text{lbl}_1^* \triangleright \text{var}_1 = \text{mod}_1, \text{lbl}_2^* \triangleright \text{var}_2 = \text{mod}_2] : \\
[\text{lbl}_1^* \triangleright \text{var}_1 : \text{sig}_1, \text{lbl}_2^* \triangleright \text{var}_2 : \text{sig}_2]
\end{array} \tag{239}$$

Rule 239: 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 (Rules 240 and 244) do not translate into explicit structure values.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longstrid}_1} \rightsquigarrow \text{path}_1 : \text{sig}_1 \quad \Gamma \vdash_{\text{ctx}} \overline{\text{longstrid}_n} \rightsquigarrow \text{path}_n : \text{sig}_n}{\Gamma \vdash \mathbf{open} \text{ longstrid}_1 \dots \text{ longstrid}_n \rightsquigarrow \text{path} : \text{sig}} \tag{240}$$

Rule 240: Though structures and datatypes both translate to structures, they have different namespaces so that *longstrid* can never specify a datatype.

$$\begin{array}{c}
\Gamma \vdash ty \rightsquigarrow con : \Omega \quad var \notin \text{bound}(\Gamma) \\
\hline
\Gamma \vdash \text{exception } id \text{ of } ty \rightsquigarrow \\
[\overline{id} \triangleright [lbl \triangleright var = \text{new_stamp}[con], mk \triangleright var \# mk, km \triangleright var \# km]] : \\
[\overline{id} \triangleright [it \triangleright var : \{mk \triangleright con \rightarrow \text{Any}, km \triangleright \text{Any} \rightarrow con\}, \\
mk \triangleright con \rightarrow \text{Any}, km \triangleright \text{Any} \rightarrow con]]]
\end{array} \quad (241)$$

Rule 241: Note that the type of *km* is partial, whereas the type of *mk* is total.

$$\begin{array}{c}
\Gamma \vdash ty \rightsquigarrow \text{Unit} : \Omega \quad var \notin \text{bound}(\Gamma) \\
\hline
\Gamma \vdash \text{exception } id \rightsquigarrow \\
[\overline{id} \triangleright [it \triangleright var = \text{new_stamp}[\text{Unit}], mk \triangleright (var \# mk) \{ \}, km \triangleright var \# km]] : \\
[\overline{id} \triangleright [it \triangleright var : \{mk \triangleright \text{Unit} \rightarrow \text{Any}, km \triangleright \text{Any} \rightarrow \text{Unit}\}, \\
mk \triangleright \text{Any}, km \triangleright \text{Any} \rightarrow \text{Unit}]]]
\end{array} \quad (242)$$

Rule 242: Nullary exceptions require special treatment, alas.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow path : sig \\
\Gamma \vdash path.mk : con_{mk} \quad \Gamma \vdash path.km : \text{Any} \rightarrow con \\
\hline
\Gamma \vdash \text{exception } id = \text{longid} \rightsquigarrow \\
[\overline{id} \triangleright [mk \triangleright path.mk, km \triangleright path.km]] : \\
[\overline{id} \triangleright [mk \triangleright con_{mk}, km \triangleright \text{Any} \rightarrow con]]]
\end{array} \quad (243)$$

Rule 243: We check that *longid* corresponds to an exception constructor (and not a datatype constructor) by looking at the type of the *km* component.

$$\begin{array}{c}
var_1 \notin \text{bound}(\Gamma) \\
\Gamma \vdash strdec_1 \rightsquigarrow mod_1 : sig_1 \\
\Gamma, lbl_1^* \triangleright var_1 : sig_1 \vdash strdec_2 \rightsquigarrow mod_2 : sig_2 \\
\Gamma, var_1 : sig_1 \vdash sig_2 \leq sig : \text{Sig} \quad \Gamma \vdash sig : \text{Sig} \\
\hline
\Gamma \vdash \text{local } strdec_1 \text{ in } strdec_2 \text{ end} \rightsquigarrow \\
[lbl_1^* \triangleright var_1 = mod_1, lbl_2 \triangleright var_2 = mod_2 : sig].lbl_2 : sig
\end{array} \quad (244)$$

Rule 244: 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.

$$\frac{\Gamma \vdash \text{tybind} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \mathbf{type} \text{ tybind} \rightsquigarrow \text{mod} : \text{sig}} \quad (245)$$

$$\frac{\Gamma \vdash \text{datbind} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \mathbf{datatype} \text{ datbind} \rightsquigarrow \text{mod} : \text{sig}} \quad (246)$$

$$\frac{\Gamma \vdash \text{strbind} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \mathbf{structure} \text{ strbind} \rightsquigarrow \text{mod} : \text{sig}} \quad (247)$$

$$\frac{\Gamma \vdash \text{funbind} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \mathbf{functor} \text{ funbind} \rightsquigarrow \text{mod} : \text{sig}} \quad (248)$$

Structure Expressions

$$\boxed{\Gamma \vdash \text{strex} \rightsquigarrow \text{mod} : \text{sig}}$$

$$\frac{\Gamma \vdash \text{strdec} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \mathbf{struct} \text{ strdec} \mathbf{end} \rightsquigarrow \text{mod} : \text{sig}} \quad (249)$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longstrid}} \rightsquigarrow \text{path} : \text{sig}}{\Gamma \vdash \text{longstrid} \rightsquigarrow \text{path} : \text{sig}} \quad (250)$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\text{ctx}} \overline{\text{funid}} \rightsquigarrow \text{path} : ((\text{var}_1 : \text{sig}_1) \rightarrow \text{sig}_2) \\ \Gamma \vdash \text{strex} \rightsquigarrow \text{mod} : \text{sig} \\ \Gamma \vdash_{\text{sub}} \text{sig} \preceq \text{sig}_1 \rightsquigarrow (\lambda \text{var} : \text{sig}. \text{mod}_1) : ((\text{var} : \text{sig}) \rightarrow \text{sig}_1'') \\ \Gamma \vdash ((\text{var}_1 : \text{sig}_1) \rightarrow \text{sig}_2) \leq (\text{sig}_1' \rightarrow \text{sig}_2') : \mathbf{Sig} \end{array}}{\Gamma \vdash \text{funid}(\text{strex}) \rightsquigarrow \text{path}((\lambda \text{var} : \text{sig}. \text{mod}_1) \text{mod}) : \text{sig}_2'} \quad (251)$$

Rule 251:

- We insert an explicit coercion to make the argument structure (which has signature sig) match the domain signature of the functor (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.

$$\begin{array}{c}
\text{var} \notin \text{bound}(\Gamma) \\
\Gamma \vdash \text{strex} \rightsquigarrow \text{mod} : \text{sig} \quad \Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig}' : \text{Sig} \\
\Gamma \vdash_{\text{sub}} \text{sig} \preceq \text{sig}' \rightsquigarrow (\lambda \text{var} : \text{sig}. \text{mod}') : ((\text{var} : \text{sig}) \rightarrow \text{sig}'') \\
\hline
\Gamma \vdash \text{strex} : \text{sigexp} \rightsquigarrow (\lambda \text{var} : \text{sig}. \text{mod}') \text{mod} : \text{sig}''
\end{array} \tag{252}$$

Rule 252: 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.

$$\begin{array}{c}
\Gamma \vdash \text{strex} \rightsquigarrow \text{mod} : \text{sig} \\
\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig}' : \text{Sig} \\
\text{var} \notin \text{bound}(\Gamma) \\
\Gamma \vdash_{\text{sub}} \text{sig} \preceq \text{sig}' \rightsquigarrow (\lambda \text{var} : \text{sig}. \text{mod}') : ((\text{var} : \text{sig}) \rightarrow \text{sig}'') \\
\hline
\Gamma \vdash \text{strex} :> \text{sigexp} \rightsquigarrow (\lambda \text{var} : \text{sig}. (\text{mod}' : \text{sig}')) \text{mod} : \text{sig}'
\end{array} \tag{253}$$

Rule 253: Ascribing a signature to a structure with $:>$ not only hides components, but restricts information about types to that which appears in the signature.

$$\begin{array}{c}
\text{var}_1 \notin \text{bound}(\Gamma) \\
\Gamma \vdash \text{strdec} \rightsquigarrow \text{mod}_1 : \text{sig}_1 \\
\Gamma, \text{lbl}_1^* \triangleright \text{var}_1 : \text{sig}_1 \vdash \text{strex} \rightsquigarrow \text{mod}_2 : \text{sig}_2 \\
\Gamma, \text{var}_1 : \text{sig}_1 \vdash \text{sig}_2 \leq \text{sig} : \text{Sig} \quad \Gamma \vdash \text{sig} : \text{Sig} \\
\hline
\Gamma \vdash \text{let strdec in strex end} \rightsquigarrow \\
[\text{lbl}_1^* \triangleright \text{var}_1 = \text{mod}_1, \text{lbl}_2 \triangleright \text{var}_2 = \text{mod}_2 : \text{sig}]. \text{lbl}_2 : \text{sig}
\end{array} \tag{254}$$

Rule 254: 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.

Structure Binding

$$\boxed{\Gamma \vdash \text{strbind} \rightsquigarrow \text{mod} : \text{sig}}$$

$$\begin{array}{c}
\Gamma \vdash \text{strex}_1 \rightsquigarrow \text{mod}_1 : \text{sig}_1 \quad \langle \dots \quad \Gamma \vdash \text{strex}_n \rightsquigarrow \text{mod}_n : \text{sig}_n \rangle \\
\hline
\Gamma \vdash \text{strid}_1 = \text{strex}_1 \langle \text{and} \dots \text{and strid}_n = \text{strex}_n \rangle \rightsquigarrow \\
[\text{strid}_1 \triangleright \text{mod}_1 \langle, \dots, \text{strid}_n \triangleright \text{mod}_n \rangle] : [\text{strid}_1 \triangleright \text{sig}_1 \langle, \dots, \text{strid}_n \triangleright \text{sig}_n \rangle]
\end{array} \tag{255}$$

Functor Binding

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

$$\frac{\begin{array}{c} \text{var} \notin \text{bound}(\Gamma) \\ \Gamma \vdash \text{sigexp}_1 \rightsquigarrow \text{sig}_1 : \text{Sig} \quad \Gamma, \overline{\text{sigid}_1} \triangleright \text{var} : \text{sig}_1 \vdash \text{strex}_1 \rightsquigarrow \text{mod}_1 : \text{sig}'_1 \\ \langle \cdot \rangle \\ \langle \Gamma \vdash \text{sigexp}_n \rightsquigarrow \text{sig}_n : \text{Sig} \quad \Gamma, \overline{\text{sigid}_n} \triangleright \text{var} : \text{sig}_n \vdash \text{strex}_n \rightsquigarrow \text{mod}_n : \text{sig}'_n \rangle \end{array}}{\Gamma \vdash \text{id}_1(\text{sigid}_1 : \text{sigexp}_1) = \text{strex}_1} \\ \frac{\langle \text{and} \ \cdots \ \text{and} \ \text{id}_n(\text{sigid}_n : \text{sigexp}_n) = \text{strex}_n \rangle \rightsquigarrow}{\frac{\overline{\text{funid}_1} \triangleright (\lambda \text{var} : \text{sig}_1. \text{mod}_1) \langle \cdot \rangle, \dots, \overline{\text{funid}_n} \triangleright (\lambda \text{var} : \text{sig}_n. \text{mod}_n) \rangle :}{\overline{\text{funid}_1} \triangleright ((\text{var} : \text{sig}_1) \rightarrow \text{sig}'_1) \langle \cdot \rangle, \dots, \overline{\text{funid}_n} \triangleright ((\text{var} : \text{sig}_n) \rightarrow \text{sig}'_n) \rangle}}{\Gamma \vdash \text{funbind} \rightsquigarrow \text{mod} : \text{sig}} \quad (256)$$

Rule 256: All functors defined explicitly in the EL have partial arrows.

Specifications

$$\boxed{\Gamma \vdash \text{spec} \rightsquigarrow [\text{sdec}s] : \text{Sig}}$$

$$\frac{\begin{array}{c} \text{FTV}(\text{ty}) = \emptyset \\ \Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega \end{array}}{\Gamma \vdash \text{val id} : \text{ty} \rightsquigarrow [\overline{\text{id}} \triangleright \text{con}] : \text{Sig}} \quad (257)$$

$$\frac{\begin{array}{c} \text{FTV}(\text{ty}) = \{\text{tyvar}_1, \dots, \text{tyvar}_n\} \neq \emptyset \\ \text{sig}_{\text{poly}} = [\overline{\text{tyvar}_1} \triangleright \Omega, \dots, \overline{\text{tyvar}_n} \triangleright \Omega] \\ \Gamma, \overline{\text{tbl}} \triangleright \text{var} : \text{sig}_{\text{poly}} \vdash \text{ty} \rightsquigarrow \text{con} : \Omega \end{array}}{\Gamma \vdash \text{val id} : \text{ty} \rightsquigarrow [\overline{\text{id}} \triangleright ((\text{var} : \text{sig}_{\text{poly}}) \rightarrow [\text{it} \triangleright \text{con}])]] : \text{Sig}} \quad (258)$$

$$\frac{\Gamma \vdash \text{typdesc} \rightsquigarrow \text{sig} : \text{Sig}}{\Gamma \vdash \text{type typdesc} \rightsquigarrow \text{sig} : \text{Sig}} \quad (259)$$

$$\frac{\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega}{\Gamma \vdash \text{exception id of ty} \rightsquigarrow [\overline{\text{id}} \triangleright [\text{mk} \triangleright \text{con} \rightarrow \text{Any}, \text{km} \triangleright \text{Any} \rightarrow \text{con}]] : \text{Sig}} \quad (260)$$

$$\frac{}{\Gamma \vdash \text{exception id} \rightsquigarrow [\overline{\text{id}} \triangleright [\text{mk} \triangleright \text{Any}, \text{km} \triangleright \text{Any} \rightarrow \text{Unit}]] : \text{Sig}} \quad (261)$$

$$\frac{\Gamma \vdash \text{datbind} \rightsquigarrow \text{mod} : \text{sig}}{\Gamma \vdash \text{datatype datbind} \rightsquigarrow \text{sig} : \text{Sig}} \quad (262)$$

$$\frac{\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}}{\Gamma \vdash \text{structure } \text{strid} : \text{sigexp} \rightsquigarrow [\overline{\text{strid}} \triangleright \text{sig}] : \text{Sig}} \quad (263)$$

$$\frac{\begin{array}{c} \Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig} \\ \text{var} \notin \text{bound}(\Gamma) \\ \Gamma, \overline{\text{strid}} \triangleright \text{var} : \text{sig} \vdash \text{sigexp}' \rightsquigarrow \text{sig}' : \text{Sig} \end{array}}{\Gamma \vdash \text{functor } \text{funid } (\text{strid} : \text{sigexp}) : \text{sigexp}' \rightsquigarrow [\overline{\text{strid}} \triangleright (\text{var} : \text{sig} \rightarrow \text{sig}')] : \text{Sig}} \quad (264)$$

$$\frac{\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}}{\Gamma \vdash \text{include } \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}} \quad (265)$$

$$\frac{\begin{array}{c} \Gamma \vdash \text{spec}_1 \rightsquigarrow [\text{sdecs}_1] : \text{Sig} \quad \Gamma, \text{sdecs} \vdash \text{spec}_2 \rightsquigarrow [\text{sdecs}_2] : \text{Sig} \\ \Gamma \vdash [\text{sdecs}_1, \text{sdecs}_2] : \text{Sig} \end{array}}{\Gamma \vdash \text{spec}_1 \text{ spec}_2 \rightsquigarrow [\text{sdecs}_1, \text{sdecs}_2] : \text{Sig}} \quad (266)$$

Rule 266: 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.

$$\frac{\begin{array}{c} \Gamma \vdash \text{spec} \rightsquigarrow \text{sig} : \text{Sig} \\ \text{var} \notin \text{bound}(\Gamma) \\ \Gamma, \text{var} : \text{sig} \vdash_{\text{sig}} \overline{\text{longid}_1} \rightsquigarrow \text{lbls}_1 : \text{knd} \\ \Gamma, \text{var} : \text{sig} \vdash_{\text{sig}} \overline{\text{longid}_2} \rightsquigarrow \text{lbls}_2 : \text{knd} \\ \Gamma, \text{var} : \text{sig} \vdash \text{var}.\text{lbls}_1 \equiv \text{var}.\text{lbls}'_1 : \text{knd} \\ \Gamma, \text{var} : \text{sig} \vdash \text{var}.\text{lbls}_2 \equiv \text{var}.\text{lbls}'_2 : \text{knd} \\ \left\{ \begin{array}{c} \text{sig} \vdash_{\text{sh}} \text{lbls}'_1 := \text{lbls}'_2 : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig} \\ \text{or} \\ \text{sig} \vdash_{\text{sh}} \text{lbls}'_2 := \text{lbls}'_1 : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig} \end{array} \right\} \end{array}}{\Gamma \vdash \text{spec } \text{sharing type } \text{longid}_1 = \text{longid}_2 \rightsquigarrow \text{sig}' : \text{Sig}} \quad (267)$$

Rule 267: 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.

Signature Expressions

$$\boxed{\Gamma \vdash sigexp \rightsquigarrow sig : \text{Sig}}$$

$$\frac{\Gamma \vdash spec \rightsquigarrow sig : \text{Sig}}{\Gamma \vdash \text{sig spec end} \rightsquigarrow sig : \text{Sig}} \quad (268)$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{sigid} \rightsquigarrow sig : \text{Sig}}{\Gamma \vdash sigid \rightsquigarrow sig : \text{Sig}} \quad (269)$$

$$\frac{\begin{array}{c} var, var_1, \dots, var_n \notin \text{bound}(\Gamma) \\ \Gamma \vdash sigexp \rightsquigarrow sig : \text{Sig} \\ \Gamma \langle \overline{tyvar_1 \triangleright var_1 : \Omega}, \dots, \overline{tyvar_n \triangleright var_n : \Omega} \rangle \vdash ty \rightsquigarrow con : \Omega \\ \Gamma, var : sig \vdash_{\text{sig}} \overline{longtycon} \rightsquigarrow lbls : \langle \Omega^n \Rightarrow \rangle \Omega \\ \Gamma, var : sig \vdash var.lbls \equiv var.lbls' : \langle \Omega^n \Rightarrow \rangle \Omega \\ sig \vdash_{\text{wt}} lbls' := \langle \lambda (var_1, \dots, var_n). \rangle con : \langle \Omega^n \Rightarrow \rangle \Omega \rightsquigarrow sig' : \text{Sig} \end{array}}{\Gamma \vdash sigexp \text{ where type } \langle (tyvar_1, \dots, tyvar_n) \rangle longtycon = ty \rightsquigarrow sig' : \text{Sig}} \quad (270)$$

Rule 270: This rule uses ambient scope for the EL type expression ty .

$$\frac{\begin{array}{c} var, var_1, \dots, var_n \notin \text{bound}(\Gamma) \\ \Gamma \vdash sigexp \rightsquigarrow sig : \text{Sig} \\ \Gamma, lbl^* \triangleright var : sig \langle \overline{tyvar_1 \triangleright var_1 : \Omega}, \dots, \overline{tyvar_n \triangleright var_n : \Omega} \rangle \vdash ty \rightsquigarrow con : \Omega \\ \Gamma, var : sig \vdash_{\text{sig}} \overline{longtycon} \rightsquigarrow lbls : \langle \Omega^n \Rightarrow \rangle \Omega \\ \Gamma, var : sig \vdash var.lbls \equiv var.lbls' : \langle \Omega^n \Rightarrow \rangle \Omega \\ sig \vdash_{\text{wt}} lbls' := \langle \lambda (var_1, \dots, var_n). \rangle con : \langle \Omega^n \Rightarrow \rangle \Omega \rightsquigarrow sig' : \text{Sig} \\ \Gamma, var : sig \vdash sig' \equiv sig'' : \text{Sig} \\ \Gamma \vdash sig'' : \text{Sig} \end{array}}{\Gamma \vdash sigexp \text{ where type } \langle (tyvar_1, \dots, tyvar_n) \rangle longtycon = \text{own } ty \rightsquigarrow sig' : \text{Sig}} \quad (271)$$

Rule 271: This rule patches signatures using internal scope for the type expression ty . As such, it may require reordering the signature components in order to satisfy scoping requirements.

$$\boxed{\Gamma \vdash topdec \rightsquigarrow tbnds : tdec}$$

$$\frac{\Gamma \vdash \text{strbind} \rightsquigarrow [sbnds] : [sdec s]}{\Gamma \vdash \text{structure strbind} \rightsquigarrow sbnds : sdec s} \quad (273)$$

$$\frac{\Gamma \vdash \text{topdec}_1 \rightsquigarrow \text{tbnds}_1 : \text{tdec}_1 \quad \Gamma, \text{tdec}_1 \vdash \text{topdec}_2 \rightsquigarrow \text{tbnds}_2 : \text{tdec}_2 \quad \Gamma \vdash \text{tdec}_1, \text{tdec}_2 \text{ ok}}{\Gamma \vdash \text{topdec}_1 \text{ topdec}_2 \rightsquigarrow \text{tbnds}_1, \text{tbnds}_2 : \text{tdec}_1, \text{tdec}_2} \quad (275)$$

Type Expressions

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

$$\frac{\Gamma \vdash ty_1 \rightsquigarrow con_1 : \Omega \quad \dots \quad \Gamma \vdash ty_n \rightsquigarrow con_n : \Omega}{\Gamma \vdash \{lab_1 : ty_1, \dots, lab_n : ty_n\} \rightsquigarrow \{\overline{lab_1} \triangleright con_1, \dots, \overline{lab_n} \triangleright con_n\} : \Omega} \quad (277)$$

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \Omega \quad \Gamma \vdash ty' \rightsquigarrow con' : \Omega}{\Gamma \vdash ty \multimap ty' \rightsquigarrow con \multimap con' : \Omega} \quad (279)$$

66

Type Definitions

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

$$\frac{\begin{array}{c} \langle \text{var}_1, \dots, \text{var}_n \notin \text{bound}(\Gamma) \rangle \\ \Gamma, \langle \overline{\text{tyvar}_1} \triangleright \text{var}_1 : \Omega, \dots, \overline{\text{tyvar}_n} \triangleright \text{var}_n : \Omega \rangle \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \\ \text{con} := \langle \lambda (\text{var}_1, \dots, \text{var}_n). \rangle \text{con} \\ \langle \langle \Gamma \vdash \text{tybind} \rightsquigarrow [sbnds] : [sdecs] \rangle \rangle \quad \langle \langle \text{var} \notin \text{FV} sdecs \rangle \rangle \end{array}}{\Gamma \vdash \langle \langle \text{tyvar}_1, \dots, \text{tyvar}_n \rangle \text{ tycon} = \text{ty} \langle \langle \text{and tybind} \rangle \rangle \rightsquigarrow} \quad (280)$$

$$\frac{\begin{array}{c} [\overline{\text{tycon}} \triangleright \text{var} = \text{con} \langle \langle, sbnds \rangle \rangle] : \\ [\overline{\text{tycon}} \triangleright \text{var} : \langle \Omega^n \Rightarrow \rangle \Omega = \text{con} \langle \langle, sdecs \rangle \rangle] \end{array}}{\Gamma \vdash \langle \langle \text{tyvar}_1, \dots, \text{tyvar}_n \rangle \text{ tycon} = \text{ty} \langle \langle \text{and tybind} \rangle \rangle \rightsquigarrow}$$

Type Descriptions

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

$$\frac{\langle \langle \Gamma \vdash \text{typdesc} \rightsquigarrow [sdecs] : \text{Sig} \rangle \rangle}{\Gamma \vdash \text{type} \langle \langle \text{tyvar}_1, \dots, \text{tyvar}_n \rangle \text{ id} \langle \langle \text{and typdesc} \rangle \rangle \rightsquigarrow} \quad (281)$$

$$[\overline{id} \triangleright \langle \Omega^n \Rightarrow \rangle \Omega \langle \langle, sdecs \rangle \rangle] : \text{Sig}$$

$$\frac{\begin{array}{c} \langle \text{var}_1, \dots, \text{var}_n \notin \text{bound}(\Gamma) \rangle \\ \Gamma, \langle \overline{\text{tyvar}_1} \triangleright \text{var}_1 : \Omega, \dots, \overline{\text{tyvar}_n} \triangleright \text{var}_n : \Omega \rangle \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \\ \text{con} := \langle \lambda (\text{var}_1, \dots, \text{var}_n). \rangle \text{con} \\ \langle \langle \Gamma \vdash \text{typdesc} \rightsquigarrow [sdecs] : \text{Sig} \rangle \rangle \quad \langle \langle \text{var} \notin \text{FV} sdecs \rangle \rangle \end{array}}{\Gamma \vdash \text{type} \langle \langle \text{tyvar}_1, \dots, \text{tyvar}_n \rangle \text{ id} = \text{ty} \langle \langle \text{and typdesc} \rangle \rangle \rightsquigarrow} \quad (282)$$

$$[\overline{id} \triangleright \langle \Omega^n \Rightarrow \rangle \Omega = \text{con} \langle \langle, sdecs \rangle \rangle] : \text{Sig}$$

Datatype Definitions

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

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

- $\text{tyvar}_1, \dots, \text{tyvar}_k$ is the union of the explicit type variables scoped in any of the datatype definitions;
- $\text{con}_i^{\text{sum}}$ is roughly the sum type representing the i^{th} datatype, with free variables $\text{var}_1^{\text{poly}}, \dots, \text{var}_k^{\text{poly}}$ representing the type variables due to polymorphism, and $\text{var}_1^{\text{dt}}, \dots, \text{var}_p^{\text{dt}}$ representing the final implementation types for the datatypes.
- The implementation type for the i^{th} datatype is $\mu_i[\text{con}^{\text{all}}]$, which still contains $\text{var}_1^{\text{poly}}, \dots, \text{var}_k^{\text{poly}}$ free.
- var^{all} (appearing in the translation) is a constructor mapping a tuple of types (the polymorphic instantiation types) to a tuple of types (the datatype implementations).
- con_i^{dt} is the implementation of the i^{th} datatype with respect to a *structure* of types var_{poly} .
- con_{ij}'' is the domain of the constructor id_{ij} , if the constructor carries a value.
- $\text{exp}_{ij}^{\text{mk}}$ is the translation of the constructor id_{ij} , and $\text{con}_{ij}^{\text{mk}}$ is its type, which is always total. These contain free variables var_{poly} and var^{all} .
- $\text{exp}_{ij}^{\text{km}}$ is the translation of the deconstructor for id_{ij} , and $\text{con}_{ij}^{\text{km}}$ is its type. The type is partial unless there is exactly one constructor in the datatype. These contain free variables var_{poly} and var^{all} .
- $\text{exp}_i^{\text{case}}$ is the non-destructuring case statement for the i^{th} datatype, and $\text{con}_i^{\text{case}}$ is its type. These contain free variables var_{poly} and var^{all} .

$$\begin{aligned}
\{tyvar_1, \dots, tyvar_k\} &= \{tyvar_{11}, \dots, tyvar_{1n_1}\} \cup \dots \cup \{tyvar_{p1}, \dots, tyvar_{pn_p}\} \\
sig_{poly} &:= [\overline{tyvar_1} \triangleright var_1^{poly} : \Omega, \dots, \overline{tyvar_k} \triangleright var_k^{poly} : \Omega] \\
sig_{poly}^+ &:= [\overline{tyvar_1} \triangleright var_1^{poly} : \Omega, \dots, \overline{tyvar_k} \triangleright var_k^{poly} : \Omega, lbl \triangleright \Omega:]
\end{aligned}$$

$$\Gamma' := \Gamma, lbl_{poly}^* \triangleright var_{poly} : sig_{poly}, \overline{tyvar_1} \triangleright var_1^{poly} : \Omega, \dots, \overline{tyvar_k} \triangleright var_k^{poly} : \Omega,$$

$$\overline{tycon_1} \triangleright var_1^{ty} : \Omega^k \Rightarrow \Omega, \dots, \overline{tycon_p} \triangleright var_p^{ty} : \Omega^k \Rightarrow \Omega$$

$$\langle con'_{ij} := [var_1^{dt} / var_1^{ty} [\dots]] \dots [var_p^{dt} / var_p^{ty} [\dots]] con_{ij} \rangle_{ij}$$

$$\langle \Gamma' \vdash ty_{ij} \rightsquigarrow con'_{ij} : \Omega \rangle_{ij}$$

$$con_i^{sum} := \Sigma \left(\left\{ \begin{array}{c} \text{Unit} \\ \text{or} \\ con'_{i1} \end{array} \right\}_{i1}, \dots, \left\{ \begin{array}{c} \text{Unit} \\ \text{or} \\ con'_{im_i} \end{array} \right\}_{im_i} \right)$$

$$con^{all} := \lambda (var_1^{dt}, \dots, var_p^{dt}). (con_1^{sum}, \dots, con_p^{sum})$$

$$con_i^{dt} := (var^{all} [(var_{poly}.tyvar_1, \dots, var_{poly}.tyvar_k)]) \# i$$

$$\langle con''_{ij} := [con_1^{dt} / var_1^{dt}] \dots [con_p^{dt} / var_p^{dt}] con'_{ij} \rangle_{ij}$$

$$exp_{ij}^{mk} := \left\{ \begin{array}{c} \text{roll}_i^{con_i^{dt}} (\text{inj}_j^{\Sigma(con''_{i1}, \dots, con''_{im_i})} \{\}) \\ \text{or} \\ \lambda var : con''_{ij}. \text{roll}_i^{con_i^{dt}} (\text{inj}_j^{\Sigma(con''_{i1}, \dots, con''_{im_i})} var) \end{array} \right\}_{ij}$$

$$con_{ij}^{mk} := \left\{ \begin{array}{c} con_i^{dt} \\ \text{or} \\ con''_{ij} \rightarrow con_i^{dt} \end{array} \right\}_{ij}$$

$$exp_{ij}^{km} := \lambda var : con_i^{dt}. \text{proj}_j (\text{unroll}_i^{con_i^{dt}} var)$$

$$con_{ij}^{km} := \left\{ \begin{array}{ll} con_i^{dt} \rightarrow con''_{ij} & \text{if } m_i = 1; \\ con_i^{dt} \rightarrow con''_{ij} & \text{otherwise.} \end{array} \right.$$

$$exp_i^{case} := \lambda var : \{1 \triangleright con_i^{dt} \rightarrow var_{poly}.lbl, \dots, m_i \triangleright con_i^{dt} \rightarrow var_{poly}.lbl\}.$$

$$\lambda var' : con_i^{dt}. \text{case}^{\Sigma(con''_{i1}, \dots, con''_{im_i})} (var \# 1, \dots, var \# m_i) \text{ of } \text{unroll}_i^{con_i^{dt}} var' \text{ end}$$

$$con_i^{case} := \{1 \triangleright con_i^{dt} \rightarrow var_{poly}.lbl, \dots, m_i \triangleright con_i^{dt} \rightarrow var_{poly}.lbl\} \rightarrow con_i^{dt} \rightarrow var_{poly}.lbl$$

$$\begin{aligned}
\Gamma \vdash & (tyvar_{11}, \dots, tyvar_{1n_1}) \ tycon_1 = id_{11} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{11} \end{array} \right\}_{11} \mid \dots \mid id_{1m_1} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{1m_1} \end{array} \right\}_{1m_1} \\
& \text{and } \dots \text{and} \\
& (tyvar_{p1}, \dots, tyvar_{pn_p}) \ tycon_p = id_{p1} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{p1} \end{array} \right\}_{p1} \mid \dots \mid id_{pm_p} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{pm_p} \end{array} \right\}_{pm_p} \rightsquigarrow \dots
\end{aligned}$$

(283)

$$\begin{aligned}
& \left[\text{lbl} \triangleright \text{var}^{all} = \lambda (var_1^{poly}, \dots, var_k^{poly}). (\mu_1 [con^{all}], \dots, \mu_p [con^{all}]), \right. \\
& \quad \overline{tycon_1}^* \triangleright \left[\overline{tycon_1} \triangleright \lambda (var_1^{poly}, \dots, var_k^{poly}). (var^{all} [(var_1^{poly}, \dots, var_k^{poly})]) \# 1, \right. \\
& \quad \quad \overline{id_{11}} \triangleright [\text{mk} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{11}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{11}^{km}])], \dots, \\
& \quad \quad \overline{id_{1m_1}} \triangleright [\text{mk} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{1m_1}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{1m_1}^{km}])], \\
& \quad \quad \text{case} \triangleright (\lambda var_{poly} : sig_{poly}^+. [\text{it} \triangleright exp_1^{case}])] \\
& \quad \vdots \\
& \quad \overline{tycon_p}^* \triangleright \left[\overline{tycon_p} \triangleright \lambda (var_1^{poly}, \dots, var_k^{poly}). (var^{all} [(var_1^{poly}, \dots, var_k^{poly})]) \# 1, \right. \\
& \quad \quad \overline{id_{p1}} \triangleright [\text{mk} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{p1}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{p1}^{km}])], \dots, \\
& \quad \quad \overline{id_{pm_p}} \triangleright [\text{mk} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{pm_p}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright (\lambda var_{poly} : sig_{poly}. [\text{it} \triangleright exp_{pm_p}^{km}])], \\
& \quad \quad \text{case} \triangleright (\lambda var_{poly} : sig_{poly}^+. [\text{it} \triangleright exp_p^{case}])] \\
& \quad \left. \right] : \\
& \left[\text{lbl} \triangleright \text{var}^{all} : \Omega^k \Rightarrow \Omega^p, \right. \\
& \quad \overline{tycon_1}^* \triangleright \left[\overline{tycon_1} \triangleright \Omega^k \Rightarrow \Omega \right. \\
& \quad \quad = \lambda (var_1^{poly}, \dots, var_k^{poly}). (var^{all} [(var_1^{poly}, \dots, var_k^{poly})]) \# 1, \\
& \quad \quad \overline{id_{11}} \triangleright [\text{mk} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{11}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{11}^{km}])], \dots, \\
& \quad \quad \overline{id_{1m_1}} \triangleright [\text{mk} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{1m_1}^{mk}]), \\
& \quad \quad \quad \text{km} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{1m_1}^{km}])], \\
& \quad \quad \text{case} \triangleright (\lambda var_{poly} : sig_{poly}^+. [\text{it} \triangleright con_1^{case}])] \\
& \quad \quad \vdots \\
& \quad \quad \overline{tycon_p}^* \triangleright \left[\overline{tycon_p} \triangleright \Omega^k \Rightarrow \Omega \right. \\
& \quad \quad \quad = \lambda (var_1^{poly}, \dots, var_k^{poly}). (var^{all} [(var_1^{poly}, \dots, var_k^{poly})]) \# 1, \\
& \quad \quad \quad \overline{id_{p1}} \triangleright [\text{mk} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{p1}^{mk}]), \\
& \quad \quad \quad \quad \text{km} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{p1}^{km}])], \dots, \\
& \quad \quad \quad \overline{id_{pm_p}} \triangleright [\text{mk} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{pm_p}^{mk}]), \\
& \quad \quad \quad \quad \text{km} \triangleright ((var_{poly} : sig_{poly}) \rightarrow [\text{it} \triangleright con_{pm_p}^{km}])], \\
& \quad \quad \quad \text{case} \triangleright (\lambda var_{poly} : sig_{poly}^+. [\text{it} \triangleright exp_p^{case}])] \\
& \quad \quad \left. \right] \\
& \left. \right]
\end{aligned}$$

8.6 Pattern Compilation

Patterns

$$\boxed{\Gamma \vdash pat \Leftarrow exp : con \mid exp' \rightsquigarrow [sdec] : [sbnd]}$$

The judgment should be read as “the bindings of the result of matching exp to the pattern pat are $[sdec] : [sbnd]$; if the pattern match fails for any reason, then exp' is evaluated.” In practice, exp' is either `fail` or `raise basis.Bind`.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{id} \not\rightsquigarrow \quad \text{or} \quad \Gamma \vdash_{\text{ctx}} \overline{id} \rightsquigarrow path : con' \quad var \notin \text{bound}(\Gamma)}{\Gamma \vdash id \Leftarrow exp : con \mid exp' \rightsquigarrow [\overline{id} \triangleright var = exp] : [\overline{id} \triangleright var : con]} \quad (284)$$

Rule 284: Pattern match against an identifier that is not a constructor. (The premise is equivalent to “rules 289 and 291 do not apply”.)

$$\frac{\begin{array}{c} lbl \text{ fresh} \quad var \notin \text{bound}(\Gamma) \\ \text{type}(scon) = con \quad \Gamma \vdash exp' : \text{Unit} \end{array}}{\Gamma \vdash scon \Leftarrow exp : con \mid exp' \rightsquigarrow [lbl \triangleright var = \text{if } exp =_{con} scon \text{ then } \{\} \text{ else } exp'] : [lbl \triangleright var : \text{Unit}]} \quad (285)$$

Rule 285: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{lbl \text{ fresh} \quad var \notin \text{bound}(\Gamma)}{\Gamma \vdash _ \Leftarrow exp : con \mid exp' \rightsquigarrow [lbl \triangleright var = exp] : [lbl \triangleright var : con]} \quad (286)$$

Rule 286: Pattern match against a wildcard.

$$\frac{\begin{array}{c} \Gamma \vdash ty \rightsquigarrow con' : \Omega \quad \Gamma \vdash con \equiv con' : \Omega \\ \Gamma \vdash pat \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig \end{array}}{\Gamma \vdash pat : ty \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig} \quad (287)$$

Rule 287: Pattern match against an explicitly-typed pattern.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : [\text{mk} \triangleright \text{con}' \rightarrow \text{con}'', \text{km} \triangleright \text{con}'' \rightarrow \text{con}'] \\
\Gamma \vdash \text{con} \equiv \text{con}'' : \Omega \quad \Gamma \vdash \text{exp}' : \text{con}' \\
\Gamma \vdash \text{pat} \Leftarrow \text{catch}(\text{path.km exp}) \text{ with } \text{exp}' : \text{con}' \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig} \\
\hline
\Gamma \vdash \text{exp}' \Leftarrow \text{exp} : \text{con} \mid \text{longid pat} \rightsquigarrow \text{mod} : \text{sig}
\end{array} \quad (288)$$

Rule 288: Pattern match against a monomorphic (datatype or exception) constructor which carries a value.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : [\text{mk} \triangleright \text{con}_{\text{mk}}, \text{km} \triangleright \text{con}'' \rightarrow \text{Unit}] \\
\Gamma \vdash \text{con} \equiv \text{con}'' : \Omega \quad \Gamma \vdash \text{exp}' : \text{Unit} \\
\text{lbl fresh} \quad \text{var} \notin \text{bound}(\Gamma) \\
\hline
\Gamma \vdash \text{longid} \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \\
[\text{lbl} \triangleright \text{var} = \text{catch path.km exp with exp}' : [\text{lbl} \triangleright \text{var} : \text{Unit}]]
\end{array} \quad (289)$$

Rule 289: Pattern match against a constant, monomorphic (datatype or exception) constructor.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : [\text{mk} \triangleright \text{sig}_{\text{mk}}, \text{km} \triangleright \text{sig}_{\text{km}}] \\
\Gamma \vdash \text{sig}_{\text{km}} \leq (\text{sig}' \rightarrow [\text{it} \triangleright \text{con}' \rightarrow \text{con}'']) : \text{Sig} \\
\Gamma \vdash \text{sig}_{\text{mk}} \leq (\text{sig}' \rightarrow [\text{it} \triangleright \text{con}'' \rightarrow \text{con}']) : \text{Sig} \\
\Gamma \vdash \text{con} \equiv \text{con}'' : \Omega \\
\Gamma \vdash \text{exp}' : \text{con}'' \\
\Gamma \vdash \text{var}' \downarrow \text{sig}' \\
\Gamma \vdash \text{pat} \Leftarrow \text{catch}((\text{path.km var}').\text{it}) \text{ exp with exp}' \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig} \\
\hline
\Gamma \vdash \text{longid pat} \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig}
\end{array} \quad (290)$$

Rule 290: Pattern match against a polymorphic datatype constructor carrying a value.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : [\text{mk} \triangleright \text{sig}_{\text{mk}}, \text{km} \triangleright \text{sig}_{\text{km}}] \\
\Gamma \vdash \text{sig}_{\text{km}} \leq (\text{sig}' \rightarrow [\text{it} \triangleright \text{con}'' \rightarrow \text{Unit}]) : \text{Sig} \\
\Gamma \vdash \text{sig}_{\text{mk}} \leq (\text{sig}' \rightarrow [\text{it} \triangleright \text{con}'']) : \text{Sig} \\
\Gamma \vdash \text{con} \equiv \text{con}'' : \Omega \\
\Gamma \vdash \text{exp}' : \text{Unit} \\
\Gamma \vdash \text{var}' \downarrow \text{sig}' \\
\hline
\Gamma \vdash \text{longid} \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \\
[\text{lbl} \triangleright \text{var} : \text{catch} ((\text{path}.\text{km} \text{var}').\text{it}) \text{exp with exp}'] : \\
[\text{lbl} \triangleright \text{var} : \text{Unit}]
\end{array} \tag{291}$$

Rule 291: Pattern match against a constant polymorphic constructor.

$$\begin{array}{c}
\text{lbl fresh} \quad \text{var} \notin \text{bound}(\Gamma) \\
\Gamma \vdash \text{con} \equiv \{ \overline{\text{lab}_1} \sim \text{con}_1, \dots, \overline{\text{lab}_n} \sim \text{con}_n, \\
\langle \text{lbl}'_1 \sim \text{con}'_1, \dots, \text{lbl}'_k \sim \text{con}'_k \rangle \} : \Omega \\
\Gamma, \text{lbl} \triangleright \text{var} : \text{con} \vdash \text{pat}_1 \Leftarrow \text{var}.\overline{\text{lab}_1} : \text{con}_1 \mid \text{exp}' \rightsquigarrow [\text{sbnds}_1] : [\text{sdecs}_1] \\
\vdots \\
\Gamma, \text{lbl} \triangleright \text{var} : \text{con} \vdash \text{pat}_n \Leftarrow \text{var}.\overline{\text{lab}_n} : \text{con}_n \mid \text{exp}' \rightsquigarrow [\text{sbnds}_n] : [\text{sdecs}_n] \\
\hline
\Gamma \vdash \{ \text{lab}_1 = \text{pat}_1, \dots, \text{lab}_n = \text{pat}_n \langle, \dots \rangle \} \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \\
[\text{lbl} \triangleright \text{var} = \text{exp}, \text{sbnds}_1, \dots, \text{sbnds}_n] : \\
[\text{lbl} \triangleright \text{var} : \text{exp}, \text{sdecs}_1, \dots, \text{sdecs}_n]
\end{array} \tag{292}$$

Rule 292: Pattern match against a record of patterns.

$$\begin{array}{c}
\text{lbl fresh} \quad \text{var} \notin \text{bound}(\Gamma) \\
\Gamma \vdash \text{pat}_1 \Leftarrow \text{var} : \text{con} \mid \text{exp}' \rightsquigarrow [\text{sbnds}_1] : [\text{sdecs}_1] \\
\Gamma \vdash \text{pat}_2 \Leftarrow \text{var} : \text{con} \mid \text{exp}' \rightsquigarrow [\text{sbnds}_2] : [\text{sdecs}_2] \\
\hline
\Gamma \vdash \text{pat}_1 \text{ as } \text{pat}_2 \Leftarrow \text{exp} : \text{con} \mid \text{exp}' \rightsquigarrow \\
[\text{lbl} \triangleright \text{var} = \text{exp}, \text{sbnds}_1, \text{sbnds}_2] : [\text{lbl} \triangleright \text{var} : \text{exp}, \text{sdecs}_1, \text{sdecs}_2]
\end{array} \tag{293}$$

Rule 293: Pattern match against two patterns simultaneously.

$$\begin{array}{c}
\Gamma \vdash \text{pat} \Leftarrow \text{get exp} : \text{con} \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig} \\
\hline
\Gamma \vdash \text{ref pat} \Leftarrow \text{exp} : \text{con} \text{ Ref} \mid \text{exp}' \rightsquigarrow \text{mod} : \text{sig}
\end{array} \tag{294}$$

Rule 294: Pattern match involving implicit dereferencing of a **ref** cell.

8.7 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 Θ 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.

Contexts

$$\boxed{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path/sig : \Theta}$$

Main rules for looking up identifiers in a translation context.

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \quad \Gamma \vdash path : con}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : con} \quad (295)$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \quad \Gamma \vdash path : knl}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : knl} \quad (296)$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \quad \Gamma \vdash path : sig}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : sig} \quad (297)$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow sig \quad \Gamma \vdash sig : \text{Sig}}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow sig : \text{Sig}} \quad (298)$$

$$\boxed{\Gamma \vdash_{\text{ctx}} lbs \rightsquigarrow path/sig}$$

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

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var:con \vdash_{\text{ctx}} lbl' \rightsquigarrow var'} \quad (299)$$

$$\frac{lbl \neq lbl' \quad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var:con \vdash_{\text{ctx}} lbl \rightsquigarrow path} \quad (300)$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var:knd \langle =con \rangle \vdash_{\text{ctx}} lbl' \rightsquigarrow var} \quad (301)$$

$$\frac{lbl \neq lbl' \quad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var:knd \langle =con \rangle \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \quad (302)$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var:sig \vdash_{\text{ctx}} lbl' \rightsquigarrow var} \quad (303)$$

$$\frac{lbl \neq lbl' \quad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var:sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \quad (304)$$

$$\frac{sig \vdash_{\text{sig}} lbl' \rightsquigarrow path}{\Gamma, lbl^* \triangleright var:sig \vdash_{\text{ctx}} lbl' \rightsquigarrow var.path} \quad (305)$$

$$\frac{sig \vdash_{\text{sig}} lbl' \not\rightsquigarrow \quad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl^* \triangleright var:sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \quad (306)$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var:\text{Sig}=sig \vdash_{\text{ctx}} lbl' \rightsquigarrow sig} \quad (307)$$

$$\frac{lbl \neq lbl' \quad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var:\text{Sig}=sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \quad (308)$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbl \rightsquigarrow path \quad sig \vdash_{\text{sig}} lbs \rightsquigarrow lbs'}{\Gamma \vdash_{\text{ctx}} lbl.lbs \rightsquigarrow path.lbs'} \quad (309)$$

Signatures

$$\boxed{\Gamma, path: sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : \Theta}$$

Main rules for looking up identifiers in a signature.

$$\frac{\Gamma \vdash path : sig \quad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \quad \Gamma \vdash path.lbls' : con}{\Gamma, path: sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : con} \quad (310)$$

$$\frac{\Gamma \vdash path : sig \quad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \quad \Gamma \vdash path.lbls' : sig'}{\Gamma, path: sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : sig'} \quad (311)$$

$$\frac{\Gamma \vdash path : sig \quad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \quad \Gamma \vdash path.lbls' : kend}{\Gamma, path: sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : kend} \quad (312)$$

$$\boxed{sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls'}$$

“Utility” rules used to look up identifiers in a signature.

$$\frac{lbl = lbl'}{[sdec, lbl \triangleright var: con] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \quad (313)$$

$$\frac{lbl \neq lbl' \quad [sdec] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdec, lbl \triangleright var: con] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \quad (314)$$

$$\frac{lbl = lbl'}{[sdec, lbl \triangleright var: kend \langle = con \rangle] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \quad (315)$$

$$\frac{lbl \neq lbl' \quad [sdec] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdec, lbl \triangleright var: kend \langle = con \rangle] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \quad (316)$$

$$\frac{lbl = lbl'}{[sdec, lbl \triangleright var: sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \quad (317)$$

$$\frac{lbl \neq lbl' \quad [sdec] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdec, lbl \triangleright var: sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \quad (318)$$

$$\frac{sig \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var: sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl^*.lbls} \quad (319)$$

$$\frac{sig \vdash_{\text{sig}} lbl' \not\rightsquigarrow \quad [sdecs] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var: sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \quad (320)$$

$$\frac{sig \vdash_{\text{ctx}} lbl \rightsquigarrow lbls' : sig' \quad sig' \vdash_{\text{sig}} path \rightsquigarrow lbls''}{sig \vdash_{\text{sig}} lbl.lbls \rightsquigarrow lbls'.lbls''} \quad (321)$$

8.8 Coercions

Matching Coercions

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

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.”

$$\frac{}{decs \vdash_{\text{sub}} sig_0 \preceq [] \rightsquigarrow (\lambda var_0: sig_0. []) : ((var_0: sig_0) \rightarrow [])} \quad (322)$$

Rule 322: Coercion to forget all components.

$$\frac{\begin{array}{l} decs, var_0: sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : con' \\ decs \vdash con \equiv con' : \Omega \\ decs, var: con \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0: sig_0. [sbnds]) : ((var_0: sig_0) \rightarrow [sdecs']) \end{array}}{decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var: con, sdecs] \rightsquigarrow \begin{array}{l} (\lambda var_0: sig_0. [lbl \triangleright var = var_0. lbls, sbnds]) : \\ ((var_0: sig_0) \rightarrow [lbl \triangleright var: con, sdecs']) \end{array}} \quad (323)$$

Rule 323: Coercion of a monomorphic value specification to a monomorphic value specification.

$$\begin{array}{c}
\left\{ \begin{array}{c} decs, var_0: sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0: sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : [mk \triangleright sig, km \triangleright sig_{km}] \end{array} \right\} \\
decs \vdash sig \leq (sig_{poly} \rightarrow [it \triangleright con]) : \text{Sig} \\
decs, var_1: sig_1 \vdash var_{poly} \downarrow sig_{poly} \\
decs, var: ((var_1: sig_1) \rightarrow [it \triangleright con]) \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\
(\lambda var_0: sig_0. [sbnds]) : ((var_0: sig_0) \rightarrow [sdecs']) \\
\hline
decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var: ((var_1: sig_1) \rightarrow [it \triangleright con]), sdecs] \rightsquigarrow \\
(\lambda var_0: sig_0. [lbl \triangleright var = (\lambda var_1: sig_1. \left\{ \begin{array}{c} var_0. lbls \\ \text{or} \\ var_0. lbls.mk \end{array} \right\} var_{poly}), sbnds]) : \\
((var_0: sig_0) \rightarrow [lbl \triangleright var: ((var_1: sig_1) \rightarrow [it \triangleright con]), sdecs'])
\end{array} \tag{324}$$

Rule 324: Coercion of a polymorphic value specification to a polymorphic value specification; this may involve implicit polymorphic instantiation. (The rule also handles matching polymorphic datatype and exception constructors with value specifications.)

$$\begin{array}{c}
\left\{ \begin{array}{c} decs, var_0: sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0: sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : [mk \triangleright sig, km \sim sig_{km}] \end{array} \right\} \\
decs \vdash sig \leq (sig_{poly} \rightarrow [it \triangleright con']) : \text{Sig} \\
decs \vdash con \equiv con' : \Omega \\
decs, var_1: sig_1 \vdash var_{poly} \downarrow sig_{poly} \\
decs, var: con \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\
(\lambda var_0: sig_0. [sbnds]) : ((var_0: sig_0) \rightarrow [sdecs']) \\
\hline
decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var: con, sdecs] \rightsquigarrow \\
(\lambda var_0: sig_0. [lbl \triangleright var = (\lambda var_1: sig_1. \left\{ \begin{array}{c} var_0. lbls \\ \text{or} \\ var_0. lbls.mk \end{array} \right\} var_{poly}).it), sbnds]) : \\
((var_0: sig_0) \rightarrow [lbl \triangleright var: con, sdecs'])
\end{array} \tag{325}$$

Rule 325: Coercion of a polymorphic value (or datatype constructor) to match a monomorphic value specification.

$$\begin{array}{c}
\text{decs}, \text{var}_0:\text{sig}_0 \vdash_{\text{sig}} \text{lbl} \rightsquigarrow \text{lbls} : \text{knd} \\
\langle \text{decs}, \text{var}_0:\text{sig}_0 \vdash \text{var}_0.\text{lbls} \equiv \text{con} : \text{knd} \rangle \\
\text{decs}, \text{var}:\text{knd}=\text{var}_0.\text{lbls} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{sbnds}]) : ((\text{var}_0:\text{sig}_0) \rightarrow [\text{sdecs}']) \\
\hline
\text{decs} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{lbl} \triangleright \text{var}:\text{knd} \langle = \text{con} \rangle, \text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{lbl} \triangleright \text{var}=\text{var}_0.\text{lbls}, \text{sbnds}]) : \\
((\text{var}_0:\text{sig}_0) \rightarrow [\text{lbl} \triangleright \text{var}:\text{knd}=\text{var}_0.\text{lbls}, \text{sdecs}'])
\end{array} \tag{326}$$

Rule 326: Coercion of a type component to a type component exposing less or equal information.

$$\begin{array}{c}
\text{decs}, \text{var}_0:\text{sig}_0 \vdash_{\text{sig}} \text{lbl} \rightsquigarrow \text{lbls} : \text{sig}_1 \\
\text{decs} \vdash_{\text{sub}} \text{sig}_1 \preceq \text{sig} \rightsquigarrow (\lambda \text{var}_1:\text{sig}_1.\text{mod}) : ((\text{var}_1:\text{sig}_1) \rightarrow \text{sig}'') \\
\text{decs}, \text{var}:\text{sig} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{sbnds}]) : ((\text{var}_0:\text{sig}_0) \rightarrow [\text{sdecs}']) \\
\hline
\text{decs} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{lbl} \triangleright \text{var}:\text{sig}, \text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{lbl} \triangleright \text{var}=(\lambda \text{var}_1:\text{sig}_1.\text{mod}) \text{var}_0.\text{lbls}, \text{sbnds}]) : \\
((\text{var}_0:\text{sig}_0) \rightarrow [\text{lbl} \triangleright \text{var}:\text{sig}'', \text{sdecs}'])
\end{array} \tag{327}$$

Rule 327: Coercion of a module component.

$$\begin{array}{c}
\text{decs} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{sdecs}'] \rightsquigarrow (\lambda \text{var}_0:\text{sig}_0.\text{mod}) : ((\text{var}_0:\text{sig}_0) \rightarrow \text{sig}') \\
\text{decs}, \text{var}:\text{sig} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{sbnds}]) : ((\text{var}_0:\text{sig}_0) \rightarrow [\text{sdecs}']) \\
\hline
\text{decs} \vdash_{\text{sub}} \text{sig}_0 \preceq [\text{lbl}^* \triangleright \text{var}:[\text{sdecs}'], \text{sdecs}] \rightsquigarrow \\
(\lambda \text{var}_0:\text{sig}_0.[\text{lbl}^* \triangleright \text{var}=\text{mod}, \text{sbnds}]) : \\
((\text{var}_0:\text{sig}_0) \rightarrow [\text{lbl}^* \triangleright \text{var}:\text{sig}', \text{sdecs}'])
\end{array} \tag{328}$$

Rule 328: Coercion to a starred substructure—this structure need not actually exist in the original structure, just the components.

$$\begin{array}{c}
\text{decs} \vdash_{\text{sub}} \text{sig}_2 \preceq \text{sig}_1 \rightsquigarrow \text{mod}_3 : \text{sig}_3 \\
\text{decs} \vdash_{\text{sub}} \text{sig}'_1 \preceq \text{sig}'_2 \rightsquigarrow \text{mod}_4 : \text{sig}_4 \\
\text{decs}, \text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1), \text{var}_2 : \text{sig}_2 \vdash \text{mod}_4 (\text{var}_0 (\text{mod}_3 \text{var}_2)) : \text{sig} \\
\hline
\text{decs} \vdash_{\text{sub}} (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1) \preceq (\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}'_2) \rightsquigarrow \\
(\lambda \text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1). (\lambda \text{var}_2 : \text{sig}_2. \text{mod}_4 (\text{var}_0 (\text{mod}_3 \text{var}_2)))) : \\
((\text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1)) \rightarrow (\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}))
\end{array} \tag{329}$$

$$\begin{array}{c}
\text{decs} \vdash_{\text{sub}} \text{sig}_2 \preceq \text{sig}_1 \rightsquigarrow \text{mod}_3 : \text{sig}_3 \\
\text{decs} \vdash_{\text{sub}} \text{sig}'_1 \preceq \text{sig}'_2 \rightsquigarrow \text{mod}_4 : \text{sig}_4 \\
\text{decs}, \text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1), \text{var}_2 : \text{sig}_2 \vdash \text{mod}_4 (\text{var}_0 (\text{mod}_3 \text{var}_2)) : \text{sig} \\
\hline
\text{decs} \vdash_{\text{sub}} (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1) \preceq (\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}'_2) \rightsquigarrow \\
(\lambda \text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1). (\lambda \text{var}_2 : \text{sig}_2. \text{mod}_4 (\text{var}_0 (\text{mod}_3 \text{var}_2)))) : \\
((\text{var}_0 : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1)) \rightarrow (\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}))
\end{array} \tag{330}$$

Rules 329, 330: Coercions for functor subtyping. By the IL subtyping rules, we can use Rule 329 to coerce a total functor to a partial functor.

8.9 Signature Patching

where type

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

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' .”

$$\frac{\begin{array}{c} \text{FV}(con) \cap \text{bound}(sdec s) = \emptyset \\ sig = [sdec s, lbl \triangleright var : knd, sdec s'] \end{array}}{sig \vdash_{\text{wt}} lbl := con : knd \rightsquigarrow [sdec s, lbl \triangleright var : knd = con, sdec s'] : \text{Sig}} \quad (331)$$

$$\frac{\begin{array}{c} \text{FV}(con) \cap \text{bound}(sdec s) = \emptyset \\ sig = [sdec s, lbl \triangleright var : sig', sdec s'] \\ sig' \vdash_{\text{wt}} lbls := con : knd \rightsquigarrow sig'' : \text{Sig} \end{array}}{sig \vdash_{\text{wt}} lbl.lbls := con : knd \rightsquigarrow [sdec s, lbl \triangleright var : sig'', sdec s'] : \text{Sig}} \quad (332)$$

sharing

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

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' .”

$$\frac{sig = [sdec s, lbl' \triangleright var' : knd, sdec s', lbl \triangleright var : knd, sdec s'']}{sig \vdash_{\text{sh}} lbl := lbl' : knd \rightsquigarrow [sdec s, lbl' \triangleright var' : knd, sdec s', lbl \triangleright var : knd = var', sdec s'] : \text{Sig}} \quad (333)$$

$$\frac{\begin{array}{c} sig = [sdec s, lbl' \triangleright var' : knd, sdec s', lbl \triangleright var : sig, sdec s''] \\ sig \vdash_{\text{wt}} lbls := var' : knd \rightsquigarrow sig' : \text{Sig} \end{array}}{sig \vdash_{\text{sh}} lbl.lbls := lbl' : knd \rightsquigarrow [sdec s, lbl' \triangleright var' : knd, sdec s', lbl \triangleright var : sig', sdec s'] : \text{Sig}} \quad (334)$$

$$\frac{\begin{array}{c} sig = [sdec s, lbl' \triangleright var' : sig', sdec s', lbl \triangleright var : sig'', sdec s''] \\ sig'' \vdash_{\text{wt}} lbls' := var'.lbls' : knd \rightsquigarrow sig''' : \text{Sig} \end{array}}{sig \vdash_{\text{sh}} lbl.lbls := lbl'.lbls' : knd \rightsquigarrow [sdec s, lbl' \triangleright var' : sig', sdec s', lbl \triangleright var : sig''', sdec s''] : \text{Sig}} \quad (335)$$

$$\frac{\begin{array}{c} sig = [sdec s, lbl \triangleright var: sig', sdec s'] \\ sig' \vdash_{sh} lbls := lbls' : knd \rightsquigarrow sig'' : \mathbf{Sig} \end{array}}{sig \vdash_{sh} lbl.lbls := lbl.lbls' : knd \rightsquigarrow [sdec s, lbl \triangleright var: sig'', sdec s'] : \mathbf{Sig}} \quad (336)$$

9 Conjectures

1. **[Type Preservation under Evaluation].**

If $\vdash tbnds : tdec$ s and $\cdot, \cdot \vdash tbnds \Downarrow \Delta, \sigma, \text{VALUES}(tbnds')$ then $\Delta \vdash tbnds' : tdec$ s.

2. **[Canonical Forms].**

<i>If a value has type/sig...</i>	<i>then it is of the form...</i>
$con_1 \rightarrow con_2$	$\text{fix } fbnds, var' = (var : con_1) \mapsto exp, fbnds' \text{ in } var' \text{ end}$
$con_1 \rightarrow con_2$	$\text{fix } fbnds, var' = (var : con_1) \mapsto exp, fbnds' \text{ in } var' \text{ end}$
$\{lbl_1 \triangleright con_1, \dots, lbl_n \triangleright con_n\}$	$\{lbl_1 \triangleright exp_{v_1}, \dots, lbl_n \triangleright exp_n\}$
$\Sigma(con_1, \dots, con_n)$	$\text{inj}_i^{\Sigma(con_1, \dots, con_n)} exp_v$
Any	$\text{tag}^{name} exp_v$
con Ref	loc
base type	$scon$
$[sdec]$	$[sbnds_v]$
$(var : sig \rightarrow sig')$	$(\lambda var : sig. mod)$
$(var : sig \rightarrow sig')$	$(\lambda var : sig. mod)$

3. **[Progress under Evaluation].** Well-typed IL programs never get “stuck” during evaluation.

4. **[Determinacy of Evaluation].** If $\cdot, \cdot \vdash tbnds \Downarrow \Delta, \sigma, \text{VALUES}(tbnds)$ and $\cdot, \cdot \vdash tbnds \Downarrow \Delta', \sigma', \text{VALUES}(tbnds')$ then $\Delta, \sigma, tbnds \equiv \Delta', \sigma', tbnds'$ up to renaming of exception names and locations bound in Δ and Δ' .

5. **[Translation Result is Well-Typed].** If $lbl^* \triangleright basis : sig_{basis} \vdash topdec \rightsquigarrow tbnds : tdec$ s then $basis : sig_{basis} \vdash tbnds : tdec$ s.

6. **[Translation Preserves Types].** Under an appropriate correspondence between EL and IL types/signatures/kinds, the translation preserves this.

References

- [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.
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [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*, 1996. To appear.

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