MIL Reference Guide (Working Draft)

Chris Stone

June 3, 2021

1 Kinds (κ)

1.1 Overview

1.1.1 Structural Rules

$$\frac{\Gamma \vdash \kappa}{\Gamma \vdash \kappa \equiv \kappa}$$

$$\frac{\Gamma \vdash \kappa' \equiv \kappa}{\Gamma \vdash \kappa \equiv \kappa'}$$

$$\frac{\Gamma \vdash \kappa \equiv \kappa' \qquad \Gamma \vdash \kappa' \equiv \kappa''}{\Gamma \vdash \kappa \equiv \kappa''}$$

$$\frac{\Gamma \vdash \kappa \equiv \kappa'}{\Gamma \vdash \kappa \preceq \kappa}$$

$$\frac{\Gamma \vdash \kappa \preceq \kappa' \qquad \Gamma \vdash \kappa' \preceq \kappa''}{\Gamma \vdash \kappa \preceq \kappa''}$$

1.2 Detailed Descriptions

1.2.1 Type kind

Description	Kind of constructors which are the types of values
Greek	T
Datatype	`Type_k`
PP	`TYPE`
Note	

Rules

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash T}$$

1.2.2 Singleton kind

Description	Kind of constructors equivalent to a given constructor
Greek	$S(c)$
Datatype	`Singleton_k(`$\langle c \rangle$`)`
PP	`SINGLE_K(`$\langle c \rangle$`)`
Note	For concision, the compiler's code actually allows singletons at all kinds. (These are definable in terms of singletons at kinds T.)

Rules

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash S(c)}$$

$$\frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash S(c) \equiv S(c')}$$

$$\frac{\Gamma \vdash S(c)}{\Gamma \vdash S(c) \preceq T} \qquad \frac{\Gamma \vdash c : S(c')}{\Gamma \vdash c \equiv c' : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash c : S(c')}$$

1.2.3 Record Kind

Description	Kinds classifying records *of* constructors (as distinguished from the type of a record value, which would have kind T).
Greek	$\{l_i \triangleright \alpha_i : \kappa_i \ ^{i \in 1..n}\}$
Datatype	`Record_k Seq[((`$\langle l_i \rangle$`,`$\langle \alpha_i \rangle$`),`$\langle \kappa_i \rangle$`)` $^{i \in 1..n}$`]`
PP	`REC_K{`$\langle l_i \rangle$`>`$\langle \alpha_i \rangle$`:`$\langle \kappa_i \rangle$ $^{i \in 1..n}$`}`
Note	The kinds of the fields can have dependencies. So for example, $$\{l_1 \triangleright \alpha : T, l_2 \triangleright \beta : S(\alpha)\}$$ is the kind classifying a record containing two constructors of kind T where the second is constrained to be equivalent to the first.

Rules

$$\frac{\Gamma \vdash \text{ok} \quad \Gamma, \alpha_i{:}\kappa_i \ ^{i<n} \vdash \kappa_n}{\Gamma \vdash \{l_i \triangleright \alpha_i{:}\kappa_i \ ^{i \in 1..n}\}}$$

$$\frac{\forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i \ ^{i<j} \vdash \kappa_j \equiv \kappa_j'}{\Gamma \vdash \{l_i \triangleright \alpha_i{:}\kappa_i \ ^{i \in 1..n}\} \equiv \{l_i \triangleright \alpha_i{:}\kappa_i' \ ^{i \in 1..n}\}}$$

$$\frac{\forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i \ ^{i<j} \vdash \kappa_j \preceq \kappa_j'}{\Gamma \vdash \{l_i \triangleright \alpha_i{:}\kappa_i \ ^{i \in 1..n}\} \preceq \{l_i \triangleright \alpha_i{:}\kappa_i' \ ^{i \in 1..n}\}}$$

1.2.4 Function Kind

Description	Kinds classifying functions mapping constructors *to* constructors (as distinguished from the type of a term-level function, which would be classified by kind T.)
Greek	$\Pi^a(\alpha_i{:}\kappa_i \ ^{i \in 1..n}).\kappa$ where $a \in \{\texttt{Open}, \texttt{Code}, \texttt{Closure}\}$
Datatype	`Arrow_k` $(\langle c \rangle$`,` `[(`$\langle \alpha_i \rangle$`,`$\langle \kappa_i \rangle$`)` $^{i \in 1..n}$`],` $\langle \kappa \rangle)$
PP	`Arrow_k(`$\langle c \rangle$`;`$\langle \alpha_i \rangle$`:`$\langle \kappa_i \rangle$ $^{i \in 1..n}$`;` $\langle \kappa \rangle$`)`
Note	Note that these are dependent function kinds.
Rules	

$$\frac{\Gamma \vdash \kappa \qquad \Gamma, \alpha_i{:}\kappa_i \ ^{i \in 1..n} \vdash \kappa}{\Gamma \vdash \Pi^a(\alpha_i{:}\kappa_i \ ^{i \in 1..n}).\kappa}$$

$$\frac{\begin{array}{c}\forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i \ ^{i<j} \vdash \kappa_j \equiv \kappa_j' \\ \Gamma, \alpha_i{:}\kappa_i \ ^{i \in 1..n} \vdash \kappa \equiv \kappa'\end{array}}{\Gamma \vdash \Pi^a(\alpha_i{:}\kappa_i \ ^{i \in 1..n}).\kappa \equiv \Pi^a(\alpha_i{:}\kappa_i' \ ^{i \in 1..n}).\kappa'}$$

$$\frac{\begin{array}{c}\forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i' \ ^{i<j} \vdash \kappa_j' \preceq \kappa_j \\ \Gamma, \alpha_i{:}\kappa_i' \ ^{i \in 1..n} \vdash \kappa \preceq \kappa'\end{array}}{\Gamma \vdash \Pi^a(\alpha_i{:}\kappa_i \ ^{i \in 1..n}).\kappa \preceq \Pi^a(\alpha_i{:}\kappa_i' \ ^{i \in 1..n}).\kappa'}$$

2 Constructors

2.1 Overview

2.1.1 Structural Rules

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c \equiv c : \kappa}$$

$$\frac{\Gamma \vdash c' \equiv c : \kappa}{\Gamma \vdash c \equiv c' : \kappa}$$

$$\frac{\Gamma \vdash c \equiv c' : \kappa \qquad \Gamma \vdash c' \equiv c'' : \kappa}{\Gamma \vdash c \equiv c'' : \kappa}$$

2.2 Detailed Descriptions

2.2.1 Integer

Description	Type of integers
Greek	Int_b where $b \in \{8, 16, 32\}$
Datatype	`Prim_c(Int_c` $\langle b \rangle$`, [])`
PP	`INT8, INT16, or INT32`
Note	The datatype contains Int_{64}, but this is not yet implemented.
Rules	

$$\frac{\Gamma \vdash \text{ok} \qquad b \in \{8, 16, 32\}}{\Gamma \vdash \mathsf{Int}_b : T}$$

2.2.2 Boxed Floating-Point

Description	Type of boxed floating-point values
Greek	$\mathsf{BoxFloat}_b$ where $b \in \{32, 64\}$
Datatype	`Prim_c(BoxFloat_c` $\langle b \rangle$`, [])`
PP	`BOXFLOAT32, or BOXFLOAT64`
Note	
Rules	

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{BoxFloat}_b : T}$$

2.2.3 Exception type

Description	Type of exception values
Greek	Exn
Datatype	`Prim_c(Exn_c, [])`
PP	`EXN`
Note	Corresponds to the Standard ML type `exn`
Rules	

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{Exn} : T}$$

2.2.4 Exception Tag

Description	Type of exception tag values
Greek	$\mathsf{ExnTag}[c]$
Datatype	`Prim_c(Exntag_c, [`$\langle c \rangle$`])`
PP	`EXNTAG(`$\langle c \rangle$`)`
Note	These are the internal tags used to distinguish exception packets. Each tag comes with the type of values it can be used to tag. (Such tagged values then have type Exn). New tags are generated at run-time when new SML `exception` constructors are created.
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathsf{ExnTag}[c] : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash \mathsf{ExnTag}[c] \equiv \mathsf{ExnTag}[c'] : T}$$

2.2.5 Array

Description	Type of array values
Greek	$\mathsf{Array}[c]$
Datatype	`Prim_c(Array_c, [`$\langle c \rangle$`])`
PP	`ARRAY(`$\langle c \rangle$`)`
Note	An array of boxed floats is represented specially; there is also a special primitive to efficiently retrieve a floating point value from such an array.
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathsf{Array}[c] : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash \mathsf{Array}[c] \equiv \mathsf{Array}[c'] : T}$$

2.2.6 Vector

Description	Type of vector values
Greek	$\mathsf{Vector}[c]$
Datatype	`Prim_c(Vector_c, [`$\langle c \rangle$`])`
PP	`VECTOR(`$\langle c \rangle$`)`
Note	Vectors are immutable arrays. Vectors of characters are strings.
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathsf{Vector}[c] : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash \mathsf{Vector}[c] \equiv \mathsf{Vector}[c'] : T}$$

2.2.7 References

Description	Type of reference values
Greek	$\mathsf{Ref}[c]$
Datatype	`Prim_c(Ref_c, [`$\langle c \rangle$`])`
PP	`REF(`$\langle c \rangle$`)`
Note	
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathsf{Ref}[c] : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash \mathsf{Ref}[c'] \equiv \mathsf{Ref}[c] : T}$$

2.2.8 Disjoint Sums

Description	Type of disjoint sum values
Greek	$\Sigma[n; c_i \ ^{i\in 1..m}]$ where $n, m \geq 0$
Datatype	`Prim_c(Sum_c{tagcount=`$\langle n\rangle$`, totalcount=`$\langle n{+}m\rangle$`, known=NONE},` `[`$\langle c_i\rangle \ ^{i\in 1..m}$`])`
PP	`SUM(`$\langle n\rangle$`)(`$\langle c_i\rangle \ ^{i\in 1..m}$`)`
Note	The layout for elements of a sum type depends on the number of non-value-carrying components. If we simply identified these with components of type `Unit`, the possible layout of a type such as `'a option` could not be determined statically (`unit option = unit+unit`, `int option = unit+int`). To avoid run-time type tests to determine layout, we statically fix a number n of non-value-carrying components, which statically determines the layout. This means that values of types such as `unit option` will be represented less compactly than is possible, but this isn't very important.
Rules	

$$\frac{\begin{array}{c} n, m \geq 0 \\ \forall i \in 1..m : \quad \Gamma \vdash c_i : T \end{array}}{\Gamma \vdash \Sigma[n; c_i \ ^{i\in 1..m}] : T}$$

$$\frac{\begin{array}{c} n, m \geq 0 \\ \forall i \in 1..m : \quad \Gamma \vdash c_i \equiv c_i' : T \end{array}}{\Gamma \vdash \Sigma[n; c_i \ ^{i\in 1..m}] \equiv \Sigma[n; c_i' \ ^{i\in 1..m}] : T}$$

2.2.9 Exposed Disjoint Sums

Description	Type of disjoint sum values whose tag is statically known.
Greek	$\Sigma_j[n; c_i\ ^{i\in 1..m}]$ where $1 \leq j \leq n+m$
Datatype	`Prim_c(Sum_c{tagcount=`$\langle n\rangle$`, totalcount=`$\langle n{+}m\rangle$`, known=SOME `$\langle j\rangle$`}, [`$\langle c_i\rangle\ ^{i\in 1..m}$`])`
PP	`SUM_`$\langle j\rangle$`(`$\langle n\rangle$`)(`$\langle c_i\rangle\ ^{i\in 1..m}$`)`
Note	
Rules	

$$\frac{j \in 1..n{+}m \qquad \forall i \in 1..m:\quad \Gamma \vdash c_i : T}{\Gamma \vdash \Sigma_j[n; c_i\ ^{i\in 1..m}] : T}$$

$$\frac{j \in 1..n{+}m \qquad \forall i \in 1..m:\quad \Gamma \vdash c_i \equiv c_i' : T}{\Gamma \vdash \Sigma_j[n; c_i\ ^{i\in 1..m}] \equiv \Sigma_j[n; c_i'\ ^{i\in 1..m}] : T}$$

2.2.10 Record

Description	Type of record values
Greek	$\{l_i{:}c_i\ ^{i\in 1..n}\}$ where $n \geq 0$, l_i distinct and sorted
Datatype	`Prim_c(Record_c [`$\langle l_i\rangle\ ^{i\in 1..n}$`], [`$\langle c_i\rangle\ ^{i\in 1..n}$`])`
PP	`RECORD[`$\langle l_i\rangle\ ^{i\in 1..n}$`](`$\langle c_i\rangle\ ^{i\in 1..n}$`)` or `UNIT` if $n = 0$
Note	
Rules	

$$\frac{\Gamma \vdash \text{ok} \qquad \forall i \in 1..n:\ \Gamma \vdash c_i : T}{\Gamma \vdash \{l_i{:}c_i\ ^{i\in 1..n}\} : T}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \forall i \in 1..n:\ \Gamma \vdash c_i \equiv c_i' : T}{\Gamma \vdash \{l_i{:}c_i\ ^{i\in 1..n}\} \equiv \{l_i{:}c_i'\ ^{i\in 1..n}\} : T}$$

2.2.11 Vararg

Description	Type constructor used in classifying `make_vararg` and `make_onearg`
Greek	$\mathsf{Vararg}^{a,e}[c, c']$ where $a \in \{\texttt{open}, \texttt{code}, \texttt{closure}\}, e \in \{\texttt{partial}, \texttt{total}\}$
Datatype	`Prim_c(Vararg_c(`$\langle a\rangle$`,`$\langle e\rangle$`), [`$\langle c\rangle$`,`$\langle c'\rangle$`])`
PP	`RECORD`$\langle a\rangle\langle e\rangle(\langle c\rangle, \langle c'\rangle)$
Note	Vararg is not implemented in TILT (yet). The pretty printer is screwed up for this case.

Rules

$$\frac{\Gamma \vdash c : T \qquad \Gamma \vdash c' : T}{\Gamma \vdash \mathsf{Vararg}^{a,e}[c, c'] : T}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : T \qquad \Gamma \vdash c_2 \equiv c_2' : T}{\Gamma \vdash \mathsf{Vararg}^{a,e}[c_1, c_2] \equiv \mathsf{Vararg}^{a,e}[c_1', c_2'] : T}$$

$$\frac{\Gamma \vdash c \equiv \{l_i{:}c_i \ ^{i \in 1..n}\} : T \qquad n \in 0..6}{\Gamma \vdash \mathsf{Vararg}^{a,e}[c, c'] \equiv \forall().(c_i \ ^{i \in 1..n}; 0) \rightarrow^{a,e} c : T}$$

2.2.12 Recursion

Description	Recursive Types
Greek	$\mu(\alpha_i{=}c_i \ ^{i \in 1..n})$ where $n \geq 0$
Datatype	`Mu_c(true, Seq[`$(\alpha_i, c_i) \ ^{i \in 1..n}$`])`
PP	`MU_C(`$\langle\alpha_i\rangle = \langle c_i\rangle \ ^{i \in 1..n}$`)` or `MU_C_NR(`$\langle\alpha_i\rangle = \langle c_i\rangle \ ^{i \in 1..n}$`)`
Note	"Tuples" of (mutually) recursively defined types; however the 1-tuple case just returns the type. The boolean flag can be set to false in the datatype if the components are not really recursive.

Rules

$$\frac{\Gamma, \alpha{:}T \vdash c : T}{\Gamma \vdash \mu(\alpha{=}c) : T} \qquad \frac{n > 1 \qquad \forall j \in 1..n :\ \Gamma, \alpha_i{:}T \ ^{i \in 1..n} \vdash c_j : T}{\Gamma \vdash \mu(\alpha_i{=}c_i \ ^{i \in 1..n}) : \{i{\triangleright}\alpha_i{:}T \ ^{i \in 1..n}\}}$$

$$\frac{\Gamma, \alpha{:}T \vdash c \equiv c' : T}{\Gamma \vdash \mu(\alpha{=}c) \equiv \mu(\alpha{=}c') : T}$$

$$\frac{n > 1 \qquad \forall j \in 1..n :\ \Gamma, \alpha_i{:}T \ ^{i \in 1..n} \vdash c_j \equiv c_j' : T}{\Gamma \vdash \mu(\alpha_i{=}c_i \ ^{i \in 1..n}) \equiv \mu(\alpha_i{=}c_i' \ ^{i \in 1..n}) : \{i{\triangleright}\alpha_i{:}T \ ^{i \in 1..n}\}}$$

2.2.13 Function

Description	Type of a term-level monomorphic function
Greek	$\forall().(c_i \ ^{i \in 1..m}; k) \rightarrow^{a,e} c$ where $a \in \{\texttt{open}, \texttt{code}, \texttt{closure}\}$ and $e \in \{\texttt{partial}, \texttt{total}\}$
Datatype	`AllArrow_c(`$\langle a \rangle$`,`$\langle e \rangle$`,[],[`$\langle c_i \rangle \ ^{i \in 1..m}$`], `$\langle k \rangle$`,`$\langle c \rangle$`)`
PP	`AllArrow(`$\langle a \rangle$`; `$\langle e \rangle$`; (); `$\langle c_i \rangle \ ^{i \in 1..m}$`; `$\langle k \rangle$`; `$\langle c \rangle$`)`
Note	The integer k specifies the number of arguments to be passed (unboxed) in floating-point registers. (There are only constructors representing monomorphic function types; there are, however, types for polymorphic functions.)
Rules	

$$\frac{\begin{array}{c} m, k \geq 0 \\ \forall j \in 1..m : \quad \Gamma \vdash c_j : T \\ \Gamma \vdash c : T \end{array}}{\Gamma \vdash \forall().(c_i \ ^{i \in 1..m}; k) \rightarrow^{a,e} c : T}$$

$$\frac{\begin{array}{c} m, k \geq 0 \\ \forall j \in 1..m : \quad \Gamma \vdash c_j \equiv c'_j : T \\ \Gamma \vdash c \equiv c' : T \end{array}}{\Gamma \vdash \forall().(c_i \ ^{i \in 1..m}; k) \rightarrow^{a,e} c \equiv \forall().(c'_i \ ^{i \in 1..m}; k) \rightarrow^{a,e} c' : T}$$

2.2.14 Type Variable

Description	Constructor variable
Greek	α
Datatype	`Var_c(`$\langle \alpha \rangle$`)`
PP	$\langle \alpha \rangle$
Note	
Rules	

$$\frac{\Gamma \vdash \mathrm{ok} \qquad \Gamma = \Gamma', \alpha{:}\kappa, \Gamma''}{\Gamma \vdash \alpha : \kappa}$$

2.2.15 Let

Description	Constructor-level let-binding
Greek	$\mathsf{let}^p \ cbnd_i \ ^{i \in 1..n} \ \mathsf{in} \ c \ \mathsf{end}$ where $p \in \{\texttt{Parallel}, \texttt{Sequential}\}$
Datatype	`Let_c (`$\langle p \rangle$`, [`$\langle cbnd_i \rangle \ ^{i \in 1..n}$`], `$\langle c \rangle$`)`

PP	`LET` $\langle cbnd_i \rangle\ ^{i \in 1..n}$ `IN` $\langle c \rangle$ `END or LETP` $\langle cbnd_i \rangle\ ^{i \in 1..n}$ `IN` $\langle c \rangle$ `END`
Note	The choices for constructor bindings are:

$$
\begin{array}{rcl}
cbnd & ::= & \alpha = c : \kappa \\
 & \mid & \alpha = \lambda^{\texttt{Open}}(\alpha_i{:}\kappa_i\ ^{i \in 1..m}){:}\kappa.c \\
 & \mid & \alpha = \lambda^{\texttt{Code}}(\alpha_i{:}\kappa_i\ ^{i \in 1..m}){:}\kappa.c
\end{array}
$$

which are represented in the datatype respectively as:

Con_cb $(\langle\alpha\rangle,\langle\kappa\rangle,\langle c\rangle)$
Open_cb $(\langle\alpha\rangle,\ [(\langle\alpha_i\rangle,\langle\kappa_i\rangle)\ ^{i \in 1..m}],\ \langle c\rangle,\langle\kappa\rangle)$
Code_cb $(\langle\alpha\rangle,\ [(\langle\alpha_i\rangle,\langle\kappa_i\rangle)\ ^{i \in 1..m}],\ \langle c\rangle,\langle\kappa\rangle)$

Notice that this serves as the intro form for constructor-level functions; all such functions are hence named. The formal rules are messy and omitted for now.

Rules

2.2.16 Constructor Record

Description	Record of constructors (as distinguished from the type of a term-level record)
Greek	$\{l_i{=}c_i\ ^{i \in 1..n}\}$
Datatype	`Crecord_c` $[(\langle l_i\rangle,\langle c_i\rangle)\ ^{i \in 1..n}]$
PP	`CREC_C`$\{\langle l_i\rangle{=}\langle c_i\rangle\ ^{i \in 1..n}\}$
Note	The order in which fields occur is significant in testing equivalence.
Rules	

$$
\frac{\forall j \in 1..n : \quad \begin{cases} \alpha_j \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash c_j : [\alpha_i{\mapsto}c_i\ ^{i<j}]\kappa_j \end{cases}}{\Gamma \vdash \{l_i{=}c_i\ ^{i \in 1..n}\} : \{l_i{\triangleright}\alpha_i{:}\kappa_i\ ^{i \in 1..n}\}}
$$

$$
\frac{\forall j \in 1..n : \quad \begin{cases} \alpha_j \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash c_j \equiv c'_i : [\alpha_i{\mapsto}c_i\ ^{i<j}]\kappa_j \end{cases}}{\Gamma \vdash \{l_i{=}c_i\ ^{i \in 1..n}\} \equiv \{l_i{=}c'_i\ ^{i \in 1..n}\} : \{l_i{\triangleright}\alpha_i{:}\kappa_i\ ^{i \in 1..n}\}}
$$

2.2.17 Projection

Description	Projection from a record of constructors
Greek	$c.l$
Datatype	`Proj_c(`$\langle c\rangle$`,`$\langle l\rangle$`)`
PP	`PROJ_C(`$\langle c\rangle$`,`$\langle l\rangle$`)`
Note	
Rules	

$$\frac{\Gamma \vdash c : \{l_i{\triangleright}\alpha_i{:}\kappa_i \ ^{i\in 1..n}\}}{\Gamma \vdash c.l_j : [\alpha_i{\mapsto}c.l_i \ ^{i<j}]\kappa_j}$$

$$\frac{\Gamma \vdash c \equiv c' : \{l_i{\triangleright}\alpha_i{:}\kappa_i \ ^{i\in 1..n}\}}{\Gamma \vdash c.l_j \equiv c'.l_j : [\alpha_i{\mapsto}c.l_i \ ^{i<j}]\kappa_j}$$

$$\frac{\Gamma \vdash \{l_i{=}c_i \ ^{i\in 1..n}\} : \{l_i{\triangleright}\alpha_i{:}\kappa_i \ ^{i\in 1..n}\} \qquad j \in 1..n}{\Gamma \vdash \{l_i{=}c_i \ ^{i\in 1..n}\}.l_j \equiv [\alpha_i{\mapsto}c_i \ ^{i<j}]c_j : [\alpha_i{\mapsto}c_i \ ^{i<j}]\kappa_j}$$

2.2.18 Closure

Description	Closure for a constructor-level function (as distinguished from the type of a term-level closure)
Greek	$\mathsf{Closure}[c_1, c_2]$
Datatype	`Closure_c (`$\langle c_1\rangle$`,`$\langle c_2\rangle$`)`
PP	`CLOSURE_C(`$\langle c_1\rangle$`,`$\langle c_2\rangle$`)`
Note	
Rules	

$$\frac{\begin{array}{c}\Gamma \vdash c : \Pi^{\texttt{code}}(\alpha_i{:}\kappa_i \ ^{i\in 1..n}).\kappa \\ \Gamma \vdash c_1 : \kappa_1\end{array}}{\Gamma \vdash \mathsf{Closure}[c, c_1] : \Pi^{\texttt{closure}}(\alpha_i{:}\{\alpha_1{\mapsto}c_1\}\kappa_i \ ^{i\in 2..n}).\{\alpha_1{\mapsto}c_1\}\kappa}$$

$$\frac{\begin{array}{c}\Gamma \vdash c \equiv c' : \Pi^{\texttt{closure}}(\alpha_i{:}\kappa_i \ ^{i\in 1..n}).\kappa \\ \Gamma \vdash c_1 \equiv c_1' : \kappa_1\end{array}}{\Gamma \vdash \mathsf{Closure}[c, c_1] \equiv \mathsf{Closure}[c', c_1'] : \Pi^{\texttt{closure}}(\alpha_i{:}\{\alpha_1{\mapsto}c_1\}\kappa_i \ ^{i\in 2..n}).\{\alpha_1{\mapsto}c_1\}\kappa}$$

2.2.19 Application

Description	Application of a constructor-level function
Greek	$c\,(c_i\ ^{i\in 1..n})$
Datatype	`App_c` ($\langle c\rangle$, [$\langle c_i\rangle$ $^{i\in 1..n}$])
PP	`APP_C` ($\langle c\rangle$, [$\langle c_i\rangle$ $^{i\in 1..n}$])
Note	Also need to write beta and eta rules.
Rules	

$$\frac{\begin{array}{c}\Gamma \vdash c : \Pi^a(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).\kappa \\ a \in \{\texttt{open}, \texttt{code}, \texttt{closure}\} \\ \forall j \in 1..n : \quad \Gamma \vdash c_j : [\alpha_i{\mapsto}c_i\ ^{i<j}]\kappa_i\end{array}}{\Gamma \vdash c\,(c_i\ ^{i\in 1..n}) : [\alpha_i{\mapsto}c_i\ ^{i\in 1..n}]\kappa}$$

$$\frac{\begin{array}{c}\Gamma \vdash c \equiv c' : \Pi^a(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).\kappa \\ a \in \{\texttt{open}, \texttt{code}, \texttt{closure}\} \\ \forall j \in 1..n : \quad \Gamma \vdash c_j \equiv c'_j : [\alpha_i{\mapsto}c_i\ ^{i<j}]\kappa_i\end{array}}{\Gamma \vdash c\,(c_i\ ^{i\in 1..n}) \equiv c'\,(c'_i\ ^{i\in 1..n}) : [\alpha_i{\mapsto}c_i\ ^{i\in 1..n}]\kappa}$$

3 Types

3.1 Overview

3.1.1 Structural Rules

$$\frac{\Gamma \vdash \kappa}{\Gamma \vdash \kappa \equiv \kappa}$$

$$\frac{\Gamma \vdash \kappa' \equiv \kappa}{\Gamma \vdash \kappa \equiv \kappa'}$$

$$\frac{\Gamma \vdash \kappa \equiv \kappa' \qquad \Gamma \vdash \kappa' \equiv \kappa''}{\Gamma \vdash \kappa \equiv \kappa''}$$

3.2 Detailed Descriptions

3.2.1 Constructors

Description	Every constructor of kind T corresponds to a type.

Greek	c
Datatype	(see above)
PP	(see above)
Note	
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash c}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : T}{\Gamma \vdash c_1 \equiv c_2}$$

3.2.2 Unboxed Floating-Point

Description	Type of unboxed floating-point values
Greek	$\mathsf{Float}_b \quad \text{where } b \in \{32, 64\}$
Datatype	`Prim_c(Float_c ⟨b⟩, [])`
PP	`FLOAT32, or FLOAT64`
Note	
Rules	

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{Float}_b}$$

3.2.3 Function

Description	Type of a term-level (possibly polymorphic) function
Greek	$\forall(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).(\tau_i\ ^{i\in 1..m}; k)\ \rightarrow^{a,e}\ \tau$ where $a \in$ {`open`, `code`, `closure`, `extern`} and $e \in$ {`partial`, `total`}
Datatype	`AllArrow_c(`$\langle a\rangle$`,`$\langle e\rangle$`,[(`$\langle\alpha_i\rangle$`,`$\langle\kappa_i\rangle$`)` $^{i\in 1..n}$`],[`$\langle\tau_i\rangle$ $^{i\in 1..m}$`], `$\langle k\rangle$`,`$\langle\tau\rangle$`)`
PP	`AllArrow(`$\langle a\rangle$`; `$\langle e\rangle$`; (`$\langle\alpha_i\rangle$` :: `$\langle\kappa_i\rangle$ $^{i\in 1..n}$`); `$\langle\tau_i\rangle$ $^{i\in 1..m}$`; `$\langle k\rangle$`; `$\langle\tau\rangle$`)`
Note	The integer k specifies the number of arguments to be passed (unboxed) in floating-point registers.
Rules	

$$\frac{\begin{array}{c} m,n,k \geq 0 \\ \forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i\ ^{i<j} \vdash \kappa_i \\ \forall j \in 1..m: \quad \Gamma, \alpha_i{:}\kappa_i\ ^{i\in 1..n} \vdash \tau_j : T \\ \Gamma, \alpha_i{:}\kappa_i\ ^{i\in 1..n} \vdash \tau : T \end{array}}{\Gamma \vdash \forall(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).(\tau_i\ ^{i\in 1..m}; k) \rightarrow^{a,e} \tau}$$

$$\frac{\begin{array}{c} m,n,k \geq 0 \\ \forall j \in 1..n: \quad \Gamma, \alpha_i{:}\kappa_i\ ^{i<j} \vdash \kappa_i \equiv \kappa_i' \\ \forall j \in 1..m: \quad \Gamma, \alpha_i{:}\kappa_i\ ^{i\in 1..n} \vdash \tau_j \equiv \tau_j' T \\ \Gamma, \alpha_i{:}\kappa_i\ ^{i\in 1..n} \vdash \tau \equiv \tau' T \end{array}}{\Gamma \vdash \forall(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).(c_i\ ^{i\in 1..m}; k) \rightarrow^{a,e} c \equiv \forall(\alpha_i{:}\kappa_i'\ ^{i\in 1..n}).(c_i'\ ^{i\in 1..m}; k) \rightarrow^{a,e} c'}$$

4 Terms

4.1 Overview

4.2 Detailed Descriptions

4.2.1 Variable

Description	Term-level variable
Greek	x
Datatype	`Var_e` $\langle x\rangle$
PP	$\langle x\rangle$
Note	
Rules	

$$\frac{\Gamma \vdash \text{ok} \qquad \Gamma = \Gamma', x{:}\tau, \Gamma''}{\Gamma \vdash x : \tau}$$

4.2.2 Constant

Description	Term-level constants
Greek	$n_b, u_b, f_b, \mathsf{vector}[c][e_i \;^{i\in 1..n}], \mathsf{array}[c][e_i \;^{i\in 1..n}], \mathsf{tag}[c](t)$
Datatype	`Const_e(int(`$\langle b\rangle$`,`$\langle n\rangle$`))`, `Const_e(uint(`$\langle b\rangle$`,`$\langle u\rangle$`))`, `Const_e(float(`$\langle b\rangle$`,`$\langle f\rangle$`))`, `Const_e(vector(`$\langle c\rangle$`,[`$\langle e_i\rangle \;^{i\in 1..n}$`]))`, `Const_e(array(`$\langle c\rangle$`,[`$\langle e_i\rangle \;^{i\in 1..n}$`]))`, `Const_e(tag(`$\langle t\rangle$`,`$\langle c\rangle$`))`
PP	$\langle n\rangle$, $\langle u\rangle$, $\langle s\rangle$, `EmptyVectorValue or VectorValue or "..."`, `ArrayValue`, `tag(`$\langle t\rangle$`,`$\langle c\rangle$`)`
Note	The constructors annotating vectors and arrays are there for the length zero case, where they cannot be inferred. In HIL there are different types for signed and unsigned integers; at the MIL level, the types are merged but there are signed and unsigned primitive operations.
Rules	

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash n_b : \mathsf{Int}_b} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash u_b : \mathsf{Int}_b} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash f_b : \mathsf{Float}_b}$$

$$\frac{\Gamma \vdash c : T \qquad \forall i \in 1..n : \quad \Gamma \vdash e_i : c}{\Gamma \vdash \mathsf{vector}[c][e_i \;^{i\in 1..n}] : \mathsf{Vector}[c]}$$

$$\frac{\Gamma \vdash c : T \qquad \forall i \in 1..n : \quad \Gamma \vdash e_i : c}{\Gamma \vdash \mathsf{array}[c][e_i \;^{i\in 1..n}] : \mathsf{Array}[c]}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \Gamma(t) = \mathsf{ExnTag}[c]}{\Gamma \vdash t : \mathsf{ExnTag}[c]}$$

4.2.3 Record

Description	Term-level record
Greek	$\{l_i{=}e_i \;^{i\in 1..n}\}$
Datatype	`Prim_e(NilPrimOp(record [`$\langle l_i\rangle \;^{i\in 1..n}$`]),[`$\langle c_i\rangle \;^{i\in 1..n}$`],[`$\langle e_i\rangle \;^{i\in 1..n}$`])`
PP	`record[`$\langle c_i\rangle \;^{i\in 1..n}$`](`$\langle e_i\rangle \;^{i\in 1..n}$`)`
Note	The `elide_prim` flag omits the constructors from pretty-printing output.
Rules	

$$\frac{\forall i \in 1..n : \quad \Gamma \vdash e_i : c_i}{\Gamma \vdash \{l_i{=}e_i \;^{i\in 1..n}\} : \{l_i{:}c_i \;^{i\in 1..n}\}}$$

4.2.4 Selection

Description	Selection from a term-level record
Greek	$e.l$
Datatype	`Prim_e(NilPrimOp(select` $\langle l_j \rangle$`,[`$\langle c \rangle$`],[`$\langle e \rangle$`]))`
PP	`select[`$\langle l_j \rangle$`]([`$\langle c \rangle$`],[`$\langle e \rangle$`])`
Note	The `elide_prim` flag omits constructor from pretty-printing output. The constructor c may be omitted, which helps avoid a blow-up in program size.
Rules	

$$\frac{\Gamma \vdash c \equiv \{l_i{:}c_i\ ^{i\in 1..n}\} : T \qquad j \in 1..n \qquad \Gamma \vdash e : c}{\Gamma \vdash e.l_j : c_j}$$

4.2.5 Sum Tagging

Description	Injection into a sum type
Greek	inj^c and $\mathsf{inj}^c\, e$ and $\mathsf{inj\_rec}^c(e_i\ ^{i\in 1..n})$
Datatype	`Prim_e(NilPrimOp(inject, [], [`$\langle e \rangle$`]))` and `Prim_e(NilPrimOp(inject_record, [], [`$\langle e_i \rangle\ ^{i\in 1..n}$`]))`
PP	`inject[`$\langle c \rangle$`](`$\langle e \rangle$`)` and `inject_rec[`$\langle c \rangle$`](`$\langle e_i \rangle\ ^{i\in 1..n}$`)`
Note	
Rules	

$$\frac{\Gamma \vdash c \equiv \Sigma_j[k; c_i\ ^{i\in 0..n-1}] : T \qquad j \in 0..(k-1)}{\Gamma \vdash \mathsf{inj}^c\ : \Sigma[k; c_i\ ^{i\in 0..n-1}]}$$

$$\frac{\Gamma \vdash c \equiv \Sigma_j[k; c_i\ ^{i\in 0..n-1}] : T \qquad j \in k..(n+k-1) \qquad \Gamma \vdash e : c_j}{\Gamma \vdash \mathsf{inj}^c\, e : \Sigma[k; c_i\ ^{i\in 0..n-1}]}$$

$$\frac{\Gamma \vdash c \equiv \Sigma_j[k; c_i\ ^{i\in 0..n-1}] : T \qquad j \in k..(n+k-1) \qquad \Gamma \vdash c_{j-k} \equiv \{l_i{:}c'_i\ ^{i\in 1..m}\} : T \qquad \forall i \in 1..m : \quad \Gamma \vdash e_i : c'_i}{\Gamma \vdash \mathsf{inj\_rec}^c(e_i\ ^{i\in 1..m}) : \Sigma[k; c_i\ ^{i\in 0..n-1}]}$$

4.2.6 Sum Untagging

Description	Projection from an exposed sum type
Greek	$\mathsf{proj}^c\, e$ and $\mathsf{proj\_rec}^c_j\, e$
Datatype	`Prim_e(NilPrimOp(project_sum, [], [`$\langle e\rangle$`]))` and `Prim_e(NilPrimOp(project_sum_record` $\langle j\rangle$`, [], [`$\langle e\rangle$`]))`
PP	`project_sum[`$\langle c\rangle$`](`$\langle e\rangle$`)` and `project_sum_rec[`$\langle j\rangle$`][`$\langle c\rangle$`](`$\langle e_i\rangle\ ^{i\in 1..n}$`)`
Note	
Rules	

$$\frac{\Gamma \vdash c \equiv \Sigma_j[k; c_i\ ^{i\in 0..n-1}] : T \qquad j \in k..(n+k-1) \quad \Gamma \vdash e : c}{\Gamma \vdash \mathsf{proj}^c\, e : c_{j-k}}$$

$$\frac{\Gamma \vdash c \equiv \Sigma_j[k; c_i\ ^{i\in 0..n-1}] : T \qquad j \in k..(n+k-1) \quad \Gamma \vdash c_{j-k} \equiv \{l_i{:}c'_i\ ^{i\in 1..m}\} : T \quad \Gamma \vdash e : c \qquad p \in 1..m}{\Gamma \vdash \mathsf{proj\_rec}^c_p\, e : c'_p}$$

4.2.7 Case analysis on Sums

Description	Case construct for sums
Greek	$\mathsf{case}^\tau\, e\, \mathsf{of}\ i \to e_i\ ^{i\in 0..k-1}, i \to (x)e_i\ ^{i\in k..k+n-1}$
Datatype	`Switch_e(Sumsw_e({sum_type=`$\langle c\rangle$`, arg=`$\langle e\rangle$`, result_type=`$\langle\tau\rangle$`, bound=`$\langle x\rangle$`, arms=[(`$\langle i\rangle$`, `$\langle e_i\rangle$`)` $^{i\in 0..k+n-1}$`], default=NONE}))`
PP	`SWITCH_SUM` $\langle c\rangle$ $\langle e\rangle$ $\langle i\rangle$`/\()()():`$\langle c'\rangle$`=`$\langle e_i\rangle\ ^{i\in 1..n}$ $\langle i\rangle$`/\()(`$\langle x\rangle$`:`$\langle c_i\rangle$`)():`$\langle c'\rangle$`=`$\langle e_i\rangle\ ^{i\in k+1..n+k-1}$ `NODEFAULT`
Note	
Rules	

$$\frac{\begin{array}{c}\Gamma \vdash e : c \\ \Gamma \vdash c \equiv \Sigma[k; c_i\ ^{i\in k+1..n+k-1}] : T \\ \forall i \in 1..k : \quad \Gamma \vdash e_i : c' \\ \forall i \in k{+}1..n{+}k{-}1 : \quad c_i := \Sigma_i[k; c_i\ ^{i\in k+1..n+k-1}] \\ \forall i \in k{+}1..n{+}k{-}1 : \quad \Gamma, x{:}c_i \vdash e_i : c'\end{array}}{\Gamma \vdash \mathsf{case}^c\, e\, \mathsf{of}\ i \to e_i\ ^{i\in 1..k}, i \to (x)e_i\ ^{i\in k+1..n+k-1} : c'}$$

4.2.8 Boxing and Unboxing

Description	Boxing and unboxing of floating-point values
Greek	$\mathsf{Box}_b\, e$ and $\mathsf{Unbox}_b\, e$ where $b \in \{32, 64\}$
Datatype	`Prim_e(NilPrimOp(box_float` $\langle b \rangle$`,[],[`$\langle e \rangle$`]))` and `Prim_e(NilPrimOp(unbox_float` $\langle b \rangle$`,[],[`$\langle e \rangle$`]))`
PP	`box_float_`$\langle b \rangle$`[](`$\langle e \rangle$`)` and `unbox_float_`$\langle b \rangle$`[](`$\langle e \rangle$`)`
Note	
Rules	

$$\frac{\Gamma \vdash e : \mathsf{Float}_b \qquad b \in \{32, 64\}}{\Gamma \vdash \mathsf{Box}_b\, e : \mathsf{BoxFloat}_b}$$

$$\frac{\Gamma \vdash e : \mathsf{BoxFloat}_b \qquad b \in \{32, 64\}}{\Gamma \vdash \mathsf{Unbox}_b\, e : \mathsf{Float}_b}$$

4.2.9 Rolling and Unrolling

Description	Coercions to and from a recursive type
Greek	$\mathsf{roll}_c\, e$ and $\mathsf{unroll}\, e$
Datatype	`Prim_e(NilPrimOp(roll,[`$\langle c \rangle$`],[`$\langle e \rangle$`]))` and `Prim_e(NilPrimOp(unroll,[],[`$\langle e \rangle$`]))`
PP	`roll[`$\langle c \rangle$`](`$\langle e \rangle$`)` and `unroll[](`$\langle e \rangle$`)`
Note	As of 2/10, `unroll` still takes the recursive type as an argument, like `roll`. This is unnecessary and we're planning to drop it.
Rules	For the $n = 1$ case:

$$\frac{\Gamma \vdash c \equiv \mu(\alpha{=}c') : T \qquad \Gamma \vdash e : \{\alpha \mapsto c\}c'}{\Gamma \vdash \mathsf{roll}_c\, e : c}$$

$$\frac{\Gamma \vdash e : \mu(\alpha{=}c)}{\Gamma \vdash \mathsf{unroll}\, e : \{\alpha \mapsto \mu(\alpha{=}c)\}c}$$

4.2.10 Exception Tags

Description	Creating a new exception tag
Greek	$\mathsf{new\_tag}[c]$
Datatype	`Prim_e(NilPrimOp(make_exntag,[`$\langle c\rangle$`],[]))`
PP	`make_exntag[`$\langle c\rangle$`]()`
Note	
Rules	

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathsf{new\_tag}[c] : \mathsf{ExnTag}[c]}$$

4.2.11 Exceptions

Description	Creating an exception value
Greek	$\mathsf{tag}\,(e_1, e_2)$
Datatype	`Prim_e(NilPrimOp(inj_exn "...",[],[`$\langle e_1\rangle$`, `$\langle e_2\rangle$`]))`
PP	`inj_exn["..."](`$\langle e_1\rangle$`,`$\langle e_2\rangle$`)`
Note	The string is used for debugging purposes.
Rules	

$$\frac{\Gamma \vdash e_1 : \mathsf{ExnTag}[c] \qquad \Gamma \vdash e_2 : c}{\Gamma \vdash \mathsf{tag}\,(e_1, e_2) : \mathsf{Exn}}$$

4.2.12 Case analysis on Exceptions

Description	Case construct for exceptions
Greek	$\mathsf{exncase}_\tau\, e\, \mathsf{of}\, e_i \to (x)e_i'\ ^{i\in 1..n}, \mathsf{default} \to e'$
Datatype	`Switch_e(Exncase_e({arg=`$\langle e\rangle$`, result_type=`$\langle \tau\rangle$`, bound=`$\langle x\rangle$`, arms=[(`$\langle e_i\rangle$`,`$\langle e_i'\rangle$`)` $^{i\in 1..n}$`], default=SOME `$\langle e'\rangle$`}))`
PP	`SWITCH_EXN `$\langle e\rangle$` `$\langle e_i\rangle$`/\()(`$\langle x\rangle$`:`$\langle c_i\rangle$`)():`$\langle c'\rangle$`=`$\langle e_i\rangle$` `$^{i\in 1..n}$` DEFAULT=`$\langle e'\rangle$
Note	
Rules	

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{Exn} \\ \forall i \in 1..n : \quad \begin{cases}\Gamma \vdash e_i : \mathsf{ExnTag}[c_i'] \\ \Gamma, x{:}c_i' \vdash e_i : c'\end{cases}\end{array}}{\Gamma \vdash (\mathsf{exncase}\, e\, \mathsf{of}\, e_i \to (x)e_i'\ ^{i\in 1..n}, \mathsf{default} \to e') : c'}$$

4.2.13 Application

Description	Application of a term-level function
Greek	$e[c_i\ ^{i\in 1..n}](e_i\ ^{i\in 1..m}; e'_i\ ^{i\in 1..p})$
Datatype	`App_e` $(\langle a\rangle, \langle e\rangle, [\langle c_i\rangle\ ^{i\in 1..n}], [\langle e_i\rangle\ ^{i\in 1..m}], [\langle e'_i\rangle\ ^{i\in 1..p}])$
PP	`App_`$\langle a\rangle(\langle e\rangle;\langle c_i\rangle\ ^{i\in 1..n};\langle e_i\rangle\ ^{i\in 1..m};\langle e'_i\rangle\ ^{i\in 1..p})$
Note	
Rules	

$$\frac{\begin{array}{c}\Gamma \vdash e : \forall(\alpha_i{:}\kappa_i\ ^{i\in 1..n}).(\tau'_i\ ^{i\in 1..m}; p) \to^{a,e} c \\ \forall j \in 1..n: \quad \Gamma \vdash c_j : [\alpha_i{\mapsto}c_i\ ^{i<j}]\kappa_j \\ \forall j \in 1..m: \quad \Gamma \vdash e_j : [\alpha_i{\mapsto}c_i\ ^{i<j}]\tau'_j \\ \forall j \in 1..p: \quad \Gamma \vdash e'_j : \mathsf{Float}_{64}\end{array}}{\Gamma \vdash e\, c_i\ ^{i\in 1..n} e_i\ ^{i\in 1..m} e'_i\ ^{i\in 1..p} : [\alpha_i{\mapsto}c_i\ ^{i\in 1..n}]\kappa}$$

4.2.14 Let

Description	Term-level let-binding
Greek	$\mathsf{let}^p\ ebnd_i\ ^{i\in 1..n}\ \mathsf{in}\ e\ \mathsf{end}$ where $p \in \{\mathtt{Parallel}, \mathtt{Sequential}\}$
Datatype	`Let_e` $(\langle p\rangle$, $[\langle ebnd_i\rangle\ ^{i\in 1..n}]$, $\langle e\rangle)$
PP	`LET` $\langle ebnd_i\rangle\ ^{i\in 1..n}$ `IN` $\langle e\rangle$ `END` or `LETP` $\langle ebnd_i\rangle\ ^{i\in 1..n}$ `IN` $\langle e\rangle$ `END`
Note	The choices for constructor bindings are:

$$
\begin{array}{rcl}
cbnd & ::= & \alpha = c \\
& \mid & x = e : \tau \\
& \mid & \{x_j = \Lambda(\alpha_{ji}{:}\kappa_{ji}\ ^{i\in 1..m_j})\lambda(x'_{ji}{:}\tau'_{ji}\ ^{i\in 1..n_j}; x''_{ji}\ ^{i\in 1..k_j})^{\mathtt{open},e_j}{:}\tau_j.e_j\ ^{j\in 1..p}\} \\
& \mid & \{x_j = \Lambda(\alpha_{ji}{:}\kappa_{ji}\ ^{i\in 1..m_j})\lambda(x'_{ji}{:}\tau'_{ji}\ ^{i\in 1..n_j}; x''_{ji}\ ^{i\in 1..k_j})^{\mathtt{code},e_j}{:}\tau_j.e_j\ ^{j\in 1..p}\} \\
& \mid & \{x_j = \mathsf{closure}(x'_j, c'_j, e_j : c_j)\ ^{j\in 1..p}\}
\end{array}
$$

which are represented in the datatype respectively as:

Con_b $(\langle\alpha\rangle,\langle\kappa\rangle,\langle c\rangle)$
Exp_b $(\langle\alpha\rangle,\langle c\rangle,\langle e\rangle)$
Fixopen_b (Set[$(\langle\alpha_j\rangle$, Function(Open, $\langle e_j\rangle$, $[(\langle\alpha_{ji}\rangle,\langle\kappa_i\rangle)\ ^{i\in 1..m}]$,
$[(\langle x'_{ji}\rangle,\langle\tau' ji\rangle)\ ^{i\in 1..n}]$, $[\langle x''_{ji}\rangle\ ^{i\in 1..k_j}]$, $\langle e_j\rangle,\langle\tau_j\rangle))\ ^{j\in 1..p}$])
Fixcode_b (Set[$(\langle\alpha_j\rangle$, Function(Code, $\langle e_j\rangle$, $[(\langle\alpha_{ji}\rangle,\langle\kappa_i\rangle)\ ^{i\in 1..m}]$,
$[(\langle x'_{ji}\rangle,\langle\tau'_{ji}\rangle)\ ^{i\in 1..n}]$, $[\langle x''_{ji}\rangle\ ^{i\in 1..k_j}]$, $\langle e_j\rangle,\langle\tau_j\rangle))\ ^{j\in 1..p}$])
Fixclosure_b (true, (Set[$(\langle x_j\rangle$, {code=$\langle x'_j\rangle$, cenv=$\langle c'_j\rangle$,
venv=$\langle e_j\rangle$, tipe=$\langle c_j\rangle$})$\ ^{j\in 1..p}$]))

Rules: Notice that this serves as the intro form for term-level functions before and after closure conversion. The formal rules are messy and omitted for now.

4.2.15 Raise and Handle

Description	Raising and handling of exceptions
Greek	$\mathsf{raise}^\tau\, e$ and $\mathsf{handle}^\tau\, e_1\, \mathsf{with}\, x \to e_2$
Datatype	`Raise_e (`$\langle e\rangle$`,`$\langle\tau\rangle$`)` and `Handle_e (`$\langle e_1\rangle$`, `$\langle x\rangle$`, `$\langle e_2\rangle$`,`$\langle\tau\rangle$`))`
PP	`RAISE(`$\langle e\rangle$`,`$\langle\tau\rangle$`)` and `HANDLE` $\langle e_1\rangle$ `:` $\langle\tau\rangle$ `WITH` $\langle x\rangle$ `: EXN .` $\langle e_2\rangle$
Note	To aid type reconstruction, raise is tagged with the type of the entire raise expression. Handle is tagged too, but there's no good reason for this
Rules	

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \mathsf{Exn}}{\Gamma \vdash \mathsf{raise}^c\, e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x{:}\mathsf{Exn} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{handle}^\tau\, e_1\, \mathsf{with}\, x \to e_2 : \tau}$$

4.2.16 Argument flattening

Description	Converting between calling conventions
Greek	$\mathsf{vararg}^{a,e}[c, c'](e)$ and $\mathsf{onearg}^{a,e}[c, c'](e)$, where $a \in \{\mathtt{open}, \mathtt{code}, \mathtt{closure}\}, e \in \{\mathtt{partial}, \mathtt{total}\}$
Datatype	`Prim_e(NilPrimOp(make_vararg(`$\langle a\rangle$`,`$\langle e\rangle$`), [`$\langle c\rangle$`,`$\langle c'\rangle$`], [`$\langle e\rangle$`]))` and `Prim_e(NilPrimOp(make_onearg(`$\langle a\rangle$`,`$\langle e\rangle$`), [`$\langle c\rangle$`,`$\langle c'\rangle$`], [`$\langle e\rangle$`]))`
PP	
Note	Currently unimplemented
Rules	

$$\frac{\Gamma \vdash e : \forall().(c; 0) \to^{a,e} c'}{\Gamma \vdash \mathsf{vararg}^{a,e}[c, c'](e) : \mathsf{Vararg}^{a,e}[c, c']}$$

$$\frac{\Gamma \vdash e : \mathsf{Vararg}^{a,e}[c, c']}{\Gamma \vdash \mathsf{onearg}^{a,e}[c, c'](e) : \forall().(c; 0) \to^{a,e} c'}$$

4.2.17 typecase

Description	Term-level typecase
Greek	$\mathsf{typecase}_\tau\, c\, \mathsf{of}\, \mathsf{BoxFloat}_{64} \to e_1, \mathsf{default} \to e_2$
Datatype	
PP	

Note — Not implemented yet. The intended implementation is more general that that shown here.

Rules

$$\frac{\Gamma \vdash c : T \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{typecase}_\tau\, c\, \mathsf{of}\ \mathsf{BoxFloat}_{64} \to e_1, \mathsf{default} \to e_2 : \tau}$$

4.2.18 trap barriers

Description — Trap barrier primitives

Greek — $\mathsf{soft\_vtrap}(tt), \mathsf{soft\_ztrap}(tt), \mathsf{hard\_vtrap}(tt), \mathsf{hard\_ztrap}(tt)$ where $tt \in \{\texttt{int\_tt}, \texttt{real\_tt}, \texttt{both\_tt}\}$

Datatype — `Prim_e(PrimOp(soft_vtrap(`$\langle tt \rangle$`)),[],[])`, `Prim_e(PrimOp(soft_ztrap(`$\langle tt \rangle$`)),[],[])`, `Prim_e(PrimOp(hard_vtrap(`$\langle tt \rangle$`)),[],[])`, `Prim_e(PrimOp(hard_ztrap(`$\langle tt \rangle$`)),[],[])`

PP — `SOFT_VTRAP, SOFT_ZTRAP, HARD_VTRAP, HARD_ZTRAP`

Note

Rules

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{soft\_vtrap}(tt) : \mathsf{Unit}} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{soft\_ztrap}(tt) : \mathsf{Unit}}$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{hard\_vtrap}(tt) : \mathsf{Unit}} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{hard\_ztrap}(tt) : \mathsf{Unit}}$$

4.2.19 reference operations

Description	Primitives for references
Greek	$\mathsf{mk\_ref}[c](e),\ \mathsf{deref}[c](e),\ \mathsf{eq\_ref}[c](e_1, e_2),\ \mathsf{setref}[c](e_1, e_2)$
Datatype	`Prim_e(PrimOp(mk_ref), [`$\langle c\rangle$`], [`$\langle e\rangle$`])`, `Prim_e(PrimOp(deref), [`$\langle c\rangle$`], [`$\langle e\rangle$`])`, `Prim_e(PrimOp(eq_ref), [`$\langle c\rangle$`], [`$\langle e_1\rangle$`,`$\langle e_2\rangle$`])`, `Prim_e(PrimOp(setref), [`$\langle c\rangle$`], [`$\langle e_1\rangle$`,`$\langle e_2\rangle$`])`
PP	
Note	Tortl details (3/31/98): The c argument for the mk_ref, deref, and setref primitives is not used if its whnf is not a path at compile-time. The c argument for eq_ref is never needed at run-time.
Rules	

$$\frac{\Gamma \vdash e : c}{\Gamma \vdash \mathsf{mk\_ref}[c](e) : \mathsf{Ref}[c]} \qquad \frac{\Gamma \vdash e : \mathsf{Ref}[c]}{\Gamma \vdash \mathsf{deref}[c](e) : c}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Ref}[c] \qquad \Gamma \vdash e_2 : c}{\Gamma \vdash \mathsf{setref}[c](e_1, e_2) : \mathsf{Unit}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Ref}[c] \qquad \Gamma \vdash e_2 : \mathsf{Ref}[c]}{\Gamma \vdash \mathsf{eq\_ref}[c](e_1, e_2) : \Sigma[2;]}$$

4.2.20 numeric conversions

Description	Primitives for numeric conversion
Greek	$\mathsf{float2int}(e),\ \mathsf{int2float}(e),\ \mathsf{int2uint}_{b_1,b_2}(e),\ \mathsf{uint2int}_{b_1,b_2}(e),\ \mathsf{int2int}_{b_1,b_2}(e),\ \mathsf{uint2uint}_{b_1,b_2}(e),\ \mathsf{uinta2uinta}_{b_1,b_2}(e),\ \mathsf{uintv2uintv}_{b_1,b_2}(e)$
Datatype	`Prim_e(PrimOp(float2int), [], [`$\langle e\rangle$`])`, `Prim_e(PrimOp(int2float), [], [`$\langle e\rangle$`])`, `Prim_e(PrimOp(int2uint (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`, `Prim_e(PrimOp(uint2int (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`, `Prim_e(PrimOp(int2int (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`, `Prim_e(PrimOp(uint2uint (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`, `Prim_e(PrimOp(uinta2uinta (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`, `Prim_e(PrimOp(uintv2uintv (`$\langle b_1\rangle$`,`$\langle b_2\rangle$`), [], [`$\langle e\rangle$`]))`

PP

Note — Since signed and unsigned integers are not distinguished at the MIL level, the typing rules for int2uint, uint2uint, and uint2int are exactly the same as the rule for int2int.

Rules

$$\frac{\Gamma \vdash e : \mathsf{Float}_{64}}{\Gamma \vdash \mathsf{float2int}(e) : \mathsf{Int}_{32}} \qquad \frac{\Gamma \vdash e : \mathsf{Int}_{32}}{\Gamma \vdash \mathsf{int2float}(e) : \mathsf{Float}_{64}}$$

$$\frac{\Gamma \vdash e : \mathsf{Int}_{b_1}}{\Gamma \vdash \mathsf{int2int}_{b_1,b_2}(e) : \mathsf{Int}_{b_2}}$$

$$\frac{\Gamma \vdash e : \mathsf{Array}[\mathsf{Int}_{b_1}]}{\Gamma \vdash \mathsf{uinta2uinta}_{b_1,b_2}(e) : \mathsf{Array}[\mathsf{Int}_{b_2}]}$$

$$\frac{\Gamma \vdash e : \mathsf{Vector}[\mathsf{Int}_{b_1}]}{\Gamma \vdash \mathsf{uintv2uintv}_{b_1,b_2}(e) : \mathsf{Vector}[\mathsf{Int}_{b_2}]}$$

4.2.21 floating-point primitives

Description — Primitives for floating-point operations

Greek — $\mathsf{neg\_float}_b(e)$, $\mathsf{abs\_float}_b(e)$, $\mathsf{plus\_float}_b(e_1, e_2)$, $\mathsf{minus\_float}_b(e_1, e_2)$, $\mathsf{mul\_float}_b(e_1, e_2)$, $\mathsf{div\_float}_b(e_1, e_2)$, $\mathsf{less\_float}_b(e_1, e_2)$, $\mathsf{greater\_float}_b(e_1, e_2)$, $\mathsf{lesseq\_float}_b(e_1, e_2)$, $\mathsf{greatereq\_float}_b(e_1, e_2)$, $\mathsf{eq\_float}_b(e_1, e_2)$, $\mathsf{neq\_float}_b(e_1, e_2)$ where $b \in \{32, 64\}$

Datatype

```
Prim_e(PrimOp(neg_float ⟨b⟩), [], [⟨e⟩]),
Prim_e(PrimOp(abs_float ⟨b⟩), [], [⟨e⟩]),
Prim_e(PrimOp(plus_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(minus_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(mul_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(less_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(greater_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(lesseq_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(greatereq_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(eq_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),
Prim_e(PrimOp(neq_float ⟨b⟩), [], [⟨e1⟩,⟨e2⟩])
```

PP

Note We give the typing rules for representative primitives only

Rules

$$\frac{\Gamma \vdash e : \mathsf{Float}_b}{\Gamma \vdash \mathsf{neg\_float}_b(e) : \mathsf{Float}_b} \qquad \frac{\Gamma \vdash e : \mathsf{Float}_b}{\Gamma \vdash \mathsf{abs\_float}_b(e) : \mathsf{Float}_b}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Float}_b \qquad \Gamma \vdash e_2 : \mathsf{Float}_b}{\Gamma \vdash \mathsf{plus\_float}_b(e_1, e_2) : \mathsf{Float}_b}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Float}_b \qquad \Gamma \vdash e_2 : \mathsf{Float}_b}{\Gamma \vdash \mathsf{less\_float}_b(e_1, e_2) : \Sigma[2;\,]}$$

4.2.22 Integer Primitives

Description	Primitives for integer operations
Greek	$\mathsf{neg\_int}_b(e)$, $\mathsf{abs\_int}_b(e)$, $\mathsf{plus\_int}_b(e_1, e_2)$, $\mathsf{minus\_int}_b(e_1, e_2)$, $\mathsf{mul\_int}_b(e_1, e_2)$, $\mathsf{div\_int}_b(e_1, e_2)$, $\mathsf{mod\_int}_b(e_1, e_2)$, $\mathsf{quot\_int}_b(e_1, e_2)$, $\mathsf{rem\_int}_b(e_1, e_2)$, $\mathsf{plus\_uint}_b(e_1, e_2)$, $\mathsf{minus\_uint}_b(e_1, e_2)$, $\mathsf{div\_uint}_b(e_1, e_2)$, $\mathsf{mod\_uint}_b(e_1, e_2)$, $\mathsf{less\_int}_b(e_1, e_2)$, $\mathsf{greater\_int}_b(e_1, e_2)$, $\mathsf{lesseq\_int}_b(e_1, e_2)$, $\mathsf{greatereq\_int}_b(e_1, e_2)$, $\mathsf{less\_uint}_b(e_1, e_2)$, $\mathsf{greater\_uint}_b(e_1, e_2)$, $\mathsf{lesseq\_uint}_b(e_1, e_2)$, $\mathsf{greatereq\_uint}_b(e_1, e_2)$, $\mathsf{eq\_int}_b(e_1, e_2)$, $\mathsf{neq\_int}_b(e_1, e_2)$, $\mathsf{not\_int}_b(e)$, $\mathsf{and\_int}_b(e_1, e_2)$, $\mathsf{or\_int}_b(e_1, e_2)$, $\mathsf{xor\_int}_b(e_1, e_2)$, $\mathsf{lshift\_int}_b(e_1, e_2)$, $\mathsf{rshift\_int}_b(e_1, e_2)$, $\mathsf{rshift\_uint}_b(e_1, e_2)$, where $b \in \{8, 16, 32\}$

Datatype	`Prim_e(PrimOp (neg_int ⟨b⟩), [], [⟨e⟩]),`
	`Prim_e(PrimOp (abs_int ⟨b⟩), [], [⟨e⟩]),`
	`Prim_e(PrimOp (plus_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (minus_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (mul_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (div_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (mod_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (quot_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (rem_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (plus_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (minus_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (mul_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (div_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (mod_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (less_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (greater_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (lesseq_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (greatereq_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (less_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (greater_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (lesseq_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (greatereq_uint ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (eq_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`
	`Prim_e(PrimOp (neq_int ⟨b⟩), [], [⟨e1⟩,⟨e2⟩]),`

PP	
Note	We show only representative typing rules. Note that although the MIL language does not distinguish in between signed and unsigned integer types, there are primitives which interpret the bit patterns as signed or unsigned and behave appropriately. (In particular, unsigned arithmetic operations do not raise exceptions.)

Rules

$$\frac{\Gamma \vdash e : \mathsf{Int}_b}{\Gamma \vdash \mathsf{neg\_int}_b(e) : \mathsf{Float}_b} \qquad \frac{\Gamma \vdash e : \mathsf{Int}_b}{\Gamma \vdash \mathsf{abs\_int}_b(e) : \mathsf{Float}_b}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_b}{\Gamma \vdash \mathsf{plus\_int}_b(e_1, e_2) : \mathsf{Int}_b} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_b}{\Gamma \vdash \mathsf{plus\_uint}_b(e_1, e_2) : \mathsf{Int}_b}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_b}{\Gamma \vdash \mathsf{less\_int}_b(e_1, e_2) : \Sigma[2;]} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_b}{\Gamma \vdash \mathsf{less\_uint}_b(e_1, e_2) : \Sigma[2;]}$$

$$\frac{\Gamma \vdash e : \mathsf{Int}_b}{\Gamma \vdash \mathsf{not\_int}_b(e) : \mathsf{Int}_b} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_b}{\Gamma \vdash \mathsf{and\_int}_b(e_1, e_2) : \mathsf{Int}_b}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_{32}}{\Gamma \vdash \mathsf{rshift\_int}_b(e_1, e_2) : \mathsf{Int}_b} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int}_b \quad \Gamma \vdash e_2 : \mathsf{Int}_{32}}{\Gamma \vdash \mathsf{rshift\_uint}_b(e_1, e_2) : \mathsf{Int}_b}$$