NIL

1. Mil (Core Nil)

1.1. Syntax.

$$
\begin{array}{rcl}
\kappa & ::= & T \mid S_T(c) \mid \Sigma(\alpha :: \kappa).\kappa \mid \Pi(\alpha :: \kappa).\kappa \\
\\
c & ::= & \alpha \mid Int \mid BoxFloat \mid \mu(a = c, b = c) \mid c \times c \mid c \to c \mid c + c \mid c\, array \\
& & \mid \lambda\alpha :: \kappa.c \mid c\, c \mid < c, c > \mid c.1 \mid c.2 \mid \mathrm{let}\, \alpha = c\, \mathrm{in}\, c\, \mathrm{end} \\
\\
\tau & ::= & T(c) \mid (\alpha_1 :: \kappa_1 \ldots \alpha_n :: \kappa_n, \tau) \to \tau \mid Float \mid \tau \times \tau \\
\\
p & ::= & \alpha \mid p.1 \mid p.2 \mid p\, c \\
\\
e & ::= & x \mid \mathrm{let}\, x = e\, \mathrm{in}\, e\, \mathrm{end} \mid \mathrm{let}\, \alpha = c\, \mathrm{in}\, e\, \mathrm{end} \\
& & \mid \mathrm{rec}\, f = \lambda(\alpha_i :: \kappa_i, x : \tau) : \tau.e \\
& & \mid e[c_1 \ldots c_n]e \mid < e, e > \mid e.1 \mid e.2 \mid n \mid r \mid boxfloat(e) \mid unboxfloat(e) \\
& & \mid inl_{c,c}e \mid inr_{c,c}e \mid case\, e\, of\, \{inl(x : c) \Rightarrow e, inr(x : c) \Rightarrow e\} \\
& & \mid \mathrm{ifBoxFloat}\, c\, \mathrm{then}\, e\, \mathrm{else}\, e \mid array_c(e, e) \mid sub(e, e) \mid fsub(e, e) \\
\\
\Delta & ::= & \bullet \mid \Delta[x : \tau] \mid \Delta[\alpha :: \kappa]
\end{array}
$$

I occasionally use $\kappa_1 \times \kappa_2$ for $\Sigma(\alpha :: \kappa_1).\kappa_2$where $\alpha \notin fv(\kappa_2)$. In places expecting sequences such as polymorphic function types or in contexts where I believe the intended meaning is clear, I abbreviate $\alpha_1 :: \kappa_1 \ldots \alpha_i :: \kappa_i$ as $\alpha_i :: \kappa_i$. So $\Delta[\alpha_i :: \kappa_i]$ means $\Delta[\alpha_1 :: \kappa_1] \ldots [\alpha_i :: \kappa_i]$ and $(\alpha_i :: \kappa_i, \tau) \to \tau$ means $(\alpha_1 :: \kappa_1 \ldots \alpha_i :: \kappa_i, \tau) \to \tau$.

1.2. The declarative system (MIL).

1.2.1. *Well Formed Context.*

Empty:

$$\frac{}{\bullet\, \mathrm{ok}}$$

Kind:

$$\frac{\Delta\, \mathrm{ok} \quad \Delta \vdash \kappa}{\Delta[\alpha :: \kappa]\, \mathrm{ok}} \quad \alpha \notin dom(\Delta)$$

Type:

$$\frac{\Delta\, \mathrm{ok} \quad \Delta \vdash \tau}{\Delta[x : \tau]\, \mathrm{ok}} \quad x \notin dom(\Delta)$$

1.2.2. *Well Formed Kind* $\Delta \vdash \kappa$.

Type:

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

Singleton:

$$\frac{\Delta \vdash c :: T}{\Delta \vdash S_T(c)}$$

Pi:

$$\frac{\Delta \vdash \kappa_1 \quad \Delta[\alpha :: \kappa_1] \vdash \kappa_2}{\Delta \vdash \Pi(\alpha :: \kappa_1).k_2}$$

Sigma:

$$\frac{\Delta \vdash \kappa_1 \quad \Delta[\alpha :: \kappa_1] \vdash \kappa_2}{\Delta \vdash \Sigma(\alpha :: \kappa_1).\kappa_2}$$

1.2.3. *Sub-Kinding* $\Delta \vdash \kappa_1 \preceq \kappa_2$.

Type:

$$\frac{\Delta \text{ ok}}{\Delta \vdash T \preceq T}$$

Singleton:

$$\frac{\Delta \vdash S_T(c)}{\Delta \vdash S_T(c) \preceq T}$$

Singletons:

$$\frac{\Delta \vdash c \equiv d :: T}{\Delta \vdash S_T(c) \preceq S_T(d)}$$

Π:

$$\frac{\Delta \vdash \kappa_1' \preceq \kappa_1 \quad \Delta[\alpha :: \kappa_1'] \vdash \kappa_2 \preceq \kappa_2'}{\Delta \vdash \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'}$$

Σ:

$$\frac{\Delta \vdash \kappa_1 \preceq \kappa_1' \quad \Delta[\alpha :: \kappa_1] \vdash \kappa_2 \preceq \kappa_2'}{\Delta \vdash \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'}$$

1.2.4. *Well-formed constructor* $\Delta \vdash c :: \kappa$.

Variable:

$$\frac{\Delta \text{ ok}}{\Delta \vdash \alpha :: \Delta(\alpha)}$$

BoxFloat:

$$\frac{\Delta \text{ ok}}{\Delta \vdash BoxFloat :: T}$$

Int:

$$\frac{\Delta \text{ ok}}{\Delta \vdash Int :: T}$$

μ:

$$\frac{\Delta[a :: T, b :: T] \vdash c_1 :: T \quad \Delta[a :: T, b :: T] \vdash c_2 :: T}{\Delta \vdash \mu(a = c_1, b = c_2) :: T}$$

Pair:

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 \times c_2 :: T}$$

Arrow:

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 \to c_2 :: T}$$

Sum:

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 + c_2 :: T}$$

Array:

$$\frac{\Delta \vdash c :: T}{\Delta \vdash c\, array :: T}$$

Lambda:

$$\frac{\Delta \vdash \kappa \quad \Delta[\alpha :: \kappa] \vdash c :: \kappa'}{\Delta \vdash \lambda(\alpha :: \kappa).c :: \Pi(\alpha :: \kappa).\kappa'}$$

App:

$$\frac{\Delta \vdash c_1 :: \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c_2 :: \kappa_1}{\Delta \vdash c_1\, c_2 :: \{c_2/\alpha\}\kappa_2}$$

Record:

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash < c_1, c_2 > :: \kappa_1 \times \kappa_2}$$

Proj1:

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \vdash c.1 :: \kappa_1}$$

Proj2:

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \vdash c.2 :: \{c.1/\alpha\}\kappa_2}$$

Let:

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta[\alpha :: \kappa_1] \vdash c_2 :: \kappa_2}{\Delta \vdash \mathrm{let}\, \alpha = c_1\, \mathrm{in}\, c_2\, \mathrm{end} :: \{c_1/\alpha\}\kappa_2}$$

Selfify:

$$\frac{\Delta \vdash c :: T}{\Delta \vdash c :: S_T(c)}$$

Subkind:

$$\frac{\Delta \vdash c :: \kappa \quad \Delta \vdash \kappa \preceq \kappa'}{\Delta \vdash c :: \kappa'}$$

Sigma Eta1:

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c.1 :: \kappa_1'}{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1').\kappa_2}$$

Sigma Eta2:

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta[\alpha :: \kappa_1] \vdash c.2 :: \kappa_2'}{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2'}$$

Pi Eta:

$$\frac{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta[\alpha :: \kappa_1] \vdash c\,\alpha :: \kappa_2'}{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2'}$$

1.2.5. *Well-formed Type* $\Delta \vdash \tau$.

Constructor:

$$\frac{\Delta \vdash c :: T}{\Delta \vdash T(c)}$$

ArrowType:

$$\frac{\begin{array}{l}\Delta[\alpha_i :: \kappa_i] \vdash \kappa_{i+1} \\ \Delta[\alpha_i :: \kappa_i] \vdash \tau_1 \qquad \Delta[\alpha_i :: \kappa_i] \vdash \tau_2\end{array}}{\Delta \vdash (\alpha_i :: \kappa_i, \tau_1) \to \tau_2}$$

Float:

$$\frac{\Delta\,\mathrm{ok}}{\Delta \vdash \mathit{Float}}$$

PairType:

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2}$$

1.2.6. *Well-typed term:* $\Delta \vdash e : \tau$.

variable:

$$\frac{\Delta\,\mathrm{ok}}{\Delta \vdash x : \Delta(x)}$$

lete:

$$\frac{\Delta \vdash e_1 : \tau_1 \quad \Delta[x : \tau_1] \vdash e_2 : \tau_2}{\Delta \vdash \mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2 \,\mathrm{end} : \tau_2}$$

letc:

$$\frac{\Delta \vdash c :: \kappa \quad \Delta[\alpha :: \kappa] \vdash e : \tau}{\Delta \vdash \mathrm{let}\, \alpha = c \,\mathrm{in}\, e \,\mathrm{end} : \{c/\alpha\}\tau}$$

rec:

$$\frac{\begin{array}{l}\Delta[\alpha_i :: \kappa_i] \vdash \kappa_{i+1} \quad \Delta[\alpha_n :: \kappa_n] \vdash \tau_1 \\ \Delta[\alpha_n :: \kappa_n] \vdash \tau_2 \quad \Delta[f : (\alpha_n :: \kappa_n, \tau_1) \to \tau_2][\alpha_n :: \kappa_n][x : \tau_1] \vdash e : \tau_2\end{array}}{\Delta \vdash \mathrm{rec}\, f = \lambda(\alpha_n :: \kappa_n, x : \tau_1) : \tau_2.e : (\alpha_n :: \kappa_n, \tau_1) \to \tau_2}$$

app:

$$\frac{\Delta \vdash e_1 : (\alpha_1 :: \kappa_1 \ldots \alpha_n :: \kappa_n, \tau_1) \to \tau_2 \quad \Delta \vdash c_{i+1} :: \{c_i/\alpha_i\}\kappa_{i+1} \quad \Delta \vdash e_2 : \{c_n/\alpha_n\}\tau_1}{\Delta \vdash e_1[c_1 \ldots c_n]e_2 : \{c_n/\alpha_n\}\tau_2}$$

pair:

$$\frac{\Delta \vdash e_1 : \tau_1 \quad \Delta \vdash e_2 : \tau_2}{\Delta \vdash < e_1, e_2 > : \tau_1 \times \tau_2}$$

proj1:

$$\frac{\Delta \vdash e : \tau_1 \times \tau_2}{\Delta \vdash e.1 : \tau_1}$$

proj2:

$$\frac{\Delta \vdash e : \tau_1 \times \tau_2}{\Delta \vdash e.2 : \tau_2}$$

float:

$$\frac{\Delta \,\mathrm{ok}}{\Delta \vdash r : Float}$$

int:

$$\frac{\Delta \,\mathrm{ok}}{\Delta \vdash n : T(Int)}$$

box:

$$\frac{\Delta \vdash e : Float}{\Delta \vdash boxfloat(e) : T(BoxFloat)}$$

unbox:

$$\frac{\Delta \vdash e : BoxFloat}{\Delta \vdash unboxfloat(e) : Float}$$

sumswitch:

$$\frac{\begin{array}{ll}\Delta \vdash c_1 :: T & \Delta[x_1 : T(c_1)] \vdash e_1 : \tau \\ \Delta \vdash c_2 :: T & \Delta[x_2 : T(c_2)] \vdash e_2 : \tau \\ \Delta \vdash e : T(c_1 + c_2) & \end{array}}{\Delta \vdash case\, e\, of\, \{inl(x_1 : c_1) \Rightarrow e_1, inr(x_2 : c_2) \Rightarrow e_2\} : \tau}$$

inl:

$$\frac{\Delta \vdash c_1 :: T \qquad \Delta \vdash c_2 :: T \quad \Delta \vdash e : T(c_1)}{\Delta \vdash inl_{c_1,c_2} e : T(c_1 + c_2)}$$

inr:

$$\frac{\Delta \vdash c_1 :: T \qquad \Delta \vdash c_2 :: T \quad \Delta \vdash e : T(c_2)}{\Delta \vdash inr_{c_1,c_2} e : T(c_1 + c_2)}$$

ifBoxFloat:

$$\frac{\Delta \vdash c :: T \quad \Delta \vdash e_1 : \tau \quad \Delta \vdash e_2 : \tau}{\Delta \vdash \text{ifBoxFloat}\, c \,\text{then}\, e_1 \,\text{else}\, e_2 : \tau}$$

array:

$$\frac{\Delta \vdash e_1 : Int \quad \Delta \vdash c :: T \quad \Delta \vdash e_2 : T(c)}{\Delta \vdash array_c(e_1, e_2) : T(c\, array)}$$

sub:

$$\frac{\Delta \vdash e_1 : T(c\, array) \quad \Delta \vdash e_2 : T(Int)}{\Delta \vdash sub(e_1, e_2) : T(c)}$$

fsub:

$$\frac{\Delta \vdash e_1 : T(BoxFloat\, array) \quad \Delta \vdash e_2 : T(Int)}{\Delta \vdash fsub(e_1, e_2) : Float}$$

2. the algorithmic MIL

2.1. **Judgments.**

2.1.1. *Well Formed Kind* $\Delta \models \kappa$.

Type:

$$\frac{}{\Delta \models T}$$

Singleton:

$$\frac{\Delta \models c \Downarrow T}{\Delta \models S_T(c)}$$

Pi:

$$\frac{\Delta \models \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models \kappa_2}{\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2}$$

Sigma:

$$\frac{\Delta \models \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models \kappa_2}{\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2}$$

2.1.2. *Sub-Kinding* $\Delta \models \kappa_1 \preceq \kappa_2$. Assume that Δ, κ_1 and κ_2 are well-formed. Check that κ_1 is a subkind of κ_2.

TS: Termination

TS: if $\Delta \vdash \kappa_1 \preceq \kappa_2$, then $\Delta \models \kappa_1 \preceq \kappa_2$

TS: if Δ ok, $\Delta \vdash \kappa_1$, $\Delta \vdash \kappa_2$, and $\Delta \models \kappa_1 \preceq \kappa_2$, then $\Delta \vdash \kappa_1 \preceq \kappa_2$

Type:

$$\frac{}{\Delta \models T \preceq T}$$

Singleton:

$$\frac{}{\Delta \models S_T(c) \preceq T}$$

Singletons:

$$\frac{\Delta \models c \equiv d :: T}{\Delta \models S_T(c) \preceq S_T(d)}$$

Π:

$$\frac{\Delta \models \kappa_1' \preceq \kappa_1 \quad \Delta[\alpha :: \kappa_1'] \models \kappa_2 \preceq \kappa_2'}{\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'} \quad \alpha \notin dom(\Delta)$$

Σ:

$$\frac{\Delta \models \kappa_1 \preceq \kappa_1' \quad \Delta[\alpha :: \kappa_1] \models \kappa_2 \preceq \kappa_2'}{\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'} \quad \alpha \notin dom(\Delta)$$

2.1.3. *Selfification* $c :: \kappa \doteq \kappa'$. This is the definition of selfification. Assume c and κ are well-formed with respect to some context. Return the most precise kind of c. Intuitively, this is the definition of a singleton at the higher kind.

TS: if $\Delta \vdash \kappa$, $\Delta \vdash c :: \kappa$, and $c :: \kappa \doteq \kappa'$, then $\Delta \vdash c :: \kappa'$.

TS: if $\Delta \vdash \kappa$, $\Delta \vdash c :: \kappa$, and $c :: \kappa \doteq \kappa'$, then for all κ'' such that $\Delta \vdash c :: \kappa''$, $\Delta \vdash \kappa' \preceq \kappa''$

Type: $c :: T \doteq S_T(c)$
Singleton: $c :: S_T(d) \doteq S_T(d)$
Π: $c :: \Pi(\alpha :: \kappa_1).\kappa_2 \doteq \Pi(\alpha :: \kappa_1).\kappa_2'$
where $c\,\alpha :: \kappa_2 \doteq \kappa_2'$
Σ: $c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \doteq \Sigma(\alpha :: \kappa_1').\kappa_2'$
where $c.1 :: \kappa_1 \doteq \kappa_1'$ and $c.2 :: \{c.1/\alpha\}\kappa_2 \doteq \kappa_2'$

2.1.4. *Kind Analysis* $\Delta \models c \Downarrow \kappa$. Assume Δ and κ are well formed. Check that c is well formed and can be given kind κ.

TS: if $\Delta \vdash c :: \kappa$ then $\Delta \models c \Downarrow \kappa$.

TS: if Δ ok, $\Delta \vdash \kappa$ and $\Delta \models c \Downarrow \kappa$, then $\Delta \vdash c :: \kappa$

$$\frac{\Delta \models c \Uparrow \kappa' \quad \Delta \models \kappa' \preceq \kappa}{\Delta \models c \Downarrow \kappa}$$

2.1.5. *Kind Synthesis* $\Delta \models c \Uparrow \kappa$. Assumes that Δ is well-formed. Check that c is well-kinded, and construct κ s.t. $\Delta \models \kappa$ and c has kind κ.

TS: if Δ ok, and $\Delta \models c \Uparrow \kappa$ then $\Delta \vdash \kappa$

TS: if Δ ok, and $\Delta \models c \Uparrow \kappa$ then $\Delta \vdash c :: \kappa$.

TS: if $\Delta \vdash c :: \kappa$, then $\Delta \models c \Uparrow \kappa'$, such that $\Delta \vdash \kappa' \preceq \kappa$

Variable:

$$\frac{\alpha :: \kappa \doteq \kappa'}{\Delta[\alpha :: \kappa] \models \alpha \Uparrow \kappa'}$$

BoxFloat:

$$\frac{}{\Delta \models BoxFloat \Uparrow S_T(BoxFloat)}$$

Int:

$$\frac{}{\Delta \models Int \Uparrow S_T(Int)}$$

μ:

$$\frac{\Delta[a :: T, b :: T] \models c_1 \Downarrow T \quad \Delta[a :: T, b :: T] \models c_2 \Downarrow T}{\Delta \models \mu(a = c_1, b = c_2) \Uparrow S_T(\mu(a = c_1, b = c_2).1) \times S_T(\mu(a = c_1, b = c_2).2)} \quad a, b \notin dom(\Delta)$$

Pair:

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 \times c_2 \Uparrow S_T(c_1 \times c_2)}$$

Arrow:

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 \to c_2 \Uparrow S_T(c_1 \to c_2)}$$

Sum:

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 + c_2 \Uparrow S_T(c_1 + c_2)}$$

Array:

$$\frac{\Delta \models c \Downarrow T}{\Delta \models c\, array \Uparrow S_T(c\, array)}$$

Lambda:

$$\frac{\Delta \models \kappa \quad \Delta[\alpha :: \kappa] \models c \Uparrow \kappa'}{\Delta \models \lambda\alpha :: \kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa'} \quad \alpha \notin dom(\Delta)$$

App:

$$\frac{\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta \models c_2 \Downarrow \kappa_1}{\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2}$$

Record:

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta \models c_2 \Uparrow \kappa_2}{\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2}$$

Proj1:

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.1 \Uparrow \kappa_1}$$

Proj2:

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.2 \Uparrow \{c.1/\alpha\}\kappa_2}$$

Let:

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2}{\Delta \models \text{let}\, \alpha = c_1\, \text{in}\, c_2\, \text{end} \Uparrow \{c_1/\alpha\}\kappa_2} \quad \alpha \notin dom(\Delta)$$

2.1.6. *Well-formed Type* $\Delta \models \tau$. Assume Δ is well-formed. Check that τ is well-formed.
TS: if Δ ok and $\Delta \models \tau$ then $\Delta \vdash \tau$.
TS: if $\Delta \vdash \tau$ then $\Delta \models \tau'$ such that $\Delta \vdash \tau \equiv \tau'$

Constructor:

$$\frac{\Delta \models c \Downarrow T}{\Delta \models T(c)}$$

ArrowType:

$$\frac{\begin{array}{l}\Delta[\alpha_i :: \kappa_i] \models \kappa_{i+1} \\ \Delta[\alpha_i :: \kappa_i] \models \tau_1 \qquad \Delta[\alpha_i :: \kappa_i] \models \tau_2\end{array}}{\Delta \models (\alpha_i :: \kappa_i, \tau_1) \to \tau_2} \quad \alpha_{i+1} \notin dom(\Delta[\alpha_i :: \kappa_i])$$

Float:

$$\frac{}{\Delta \models Float}$$

PairType:

$$\frac{\Delta \models \tau_1 \quad \Delta \models \tau_2}{\Delta \models \tau_1 \times \tau_2}$$

2.1.7. *Type Analysis* $\Delta \models e \Downarrow \tau$. Assume Δ and τ are well-formed. Check that e is well-typed, and has type τ.
TS: if Δ ok, $\Delta \vdash \tau$ and $\Delta \models e \Downarrow \tau$, then $\Delta \vdash e : \tau$
TS: if $\Delta \vdash e : \tau$ then $\Delta \models e \Downarrow \tau$

$$\frac{\Delta \models e \Uparrow \tau' \quad \Delta \models \tau' \equiv \tau}{\Delta \models e \Downarrow \tau}$$

2.1.8. *Type Synthesis* $\Delta \models e \Uparrow \tau$. Assume Δ is well-formed. Check that e is well-formed and construct its type τ, where $\Delta \models \tau$
TS: if Δ ok and $\Delta \models e \Uparrow \tau$ then $\Delta \vdash \tau$.
TS: if Δ ok and $\Delta \models e \Uparrow \tau$ then $\Delta \vdash e : \tau$
TS if $\Delta \vdash e : \tau$, then $\Delta \models e \Uparrow \tau'$ such that $\Delta \vdash \tau \equiv \tau'$

variable:

$$\frac{}{\Delta[x : \tau] \models x \Uparrow \tau}$$

lete:

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta[x : \tau_1] \models e_2 \Uparrow \tau_2}{\Delta \models \text{let } x = e_1 \text{ in } e_2 \text{ end} \Uparrow \tau_2} \quad x \notin dom(\Delta)$$

letc:

$$\frac{\Delta \models c \Uparrow \kappa \quad \Delta[\alpha :: \kappa] \models e \Uparrow \tau}{\Delta \models \text{let } \alpha = c \text{ in } e \text{ end} \Uparrow \{c/\alpha\}\tau} \quad \alpha \notin dom(\Delta)$$

rec:

$$\frac{\begin{array}{l}\Delta[\alpha_i :: \kappa_i] \models \kappa_{i+1} \quad \Delta[\alpha_n :: \kappa_n] \models \tau_1 \\ \Delta[\alpha_n :: \kappa_n] \models \tau_2 \quad \Delta[f : (\alpha_n :: \kappa_n, \tau_1) \to \tau_2][\alpha_n :: \kappa_n][x : \tau_1] \models e \Downarrow \tau_2\end{array}}{\Delta \models \text{rec } f = \lambda(\alpha_n :: \kappa_n, x : \tau_1) : \tau_2.e \Uparrow (\alpha_n :: \kappa_n, \tau_1) \to \tau_2}$$

where $f, x \notin dom(\Delta), \alpha_{i+1} \notin dom(\Delta[\alpha_i :: \kappa_i]$.

app:

$$\frac{\Delta \models e_1 \Uparrow (\alpha_1 :: \kappa_1 \ldots \alpha_n :: \kappa_n, \tau_1) \to \tau_2 \quad \Delta \models c_{i+1} \Downarrow \{c_i/\alpha_i\}\kappa_{i+1} \quad \Delta \models e_2 \Downarrow \{c_n/\alpha_n\}\tau_1}{\Delta \models e_1[c_1 \ldots c_n]e_2 \Uparrow \{c_n/\alpha_n\}\tau_2}$$

mono_con_app:

$$\frac{\Delta \models e_1 \Uparrow T(c_e) \quad \Delta \models c_e \Longrightarrow c_1 \to c_2 \quad \Delta \models e_2 \Downarrow T(c_1)}{\Delta \models e_1[]e_2 \Uparrow T(c_2)}$$

pair:

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta \models e_2 \Uparrow \tau_2}{\Delta \models < e_1, e_2 > \Uparrow \tau_1 \times \tau_2}$$

type_proj1:

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.1 \Uparrow \tau_1}$$

con_proj1:

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \Longrightarrow c_1 \times c_2}{\Delta \models e.1 \Uparrow T(c_1)}$$

type_proj2:

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.2 \Uparrow \tau_2}$$

con_proj2:

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \Longrightarrow c_1 \times c_2}{\Delta \models e.2 \Uparrow T(c_2)}$$

float:

$$\frac{}{\Delta \models r \Uparrow Float}$$

int:

$$\frac{}{\Delta \models n \Uparrow T(Int)}$$

box:

$$\frac{\Delta \models e \Downarrow Float}{\Delta \models boxfloat(e) \Uparrow T(BoxFloat)}$$

unbox:

$$\frac{\Delta \models e \Downarrow T(BoxFloat)}{\Delta \models unboxfloat(e) \Uparrow Float}$$

sumswitch:

$$\frac{\begin{array}{ll} \Delta \models c_1 \Downarrow T & \Delta[x_1 : T(c_1)] \models e_1 \Uparrow \tau_1 \\ \Delta \models c_2 \Downarrow T & \Delta[x_2 : T(c_2)] \models e_2 \Uparrow \tau_2 \\ \Delta \models e \Downarrow T(c_1 + c_2) & \Delta \models \tau_1 \equiv \tau_2 \end{array}}{\Delta \models case\, e\, of\, \{inl(x_1 : c_1) \Rightarrow e_1, inr(x_2 : c_2) \Rightarrow e_2\} \Uparrow \tau_1}$$

inl:

$$\frac{\Delta \models c_1 \Downarrow T \qquad \Delta \models c_2 \Downarrow T \quad \Delta \models e \Downarrow T(c_1)}{\Delta \models inl_{c_1,c_2} e \Uparrow T(c_1 + c_2)}$$

inr:

$$\frac{\Delta \models c \Downarrow T \qquad \Delta \models c_2 \Downarrow T \quad \Delta \models e \Downarrow T(c_2)}{\Delta \models inr_{c_1,c_2} e \Uparrow T(c_1 + c_2)}$$

ifBoxFloat:

$$\frac{\Delta \models c \Downarrow T \quad \Delta \models e_1 \Uparrow \tau_1 \quad \Delta \models e_2 \Uparrow \tau_2 \quad \Delta \models \tau_1 \equiv \tau_2}{\Delta \models \text{ifBoxFloat}\, c \,\text{then}\, e_1 \,\text{else}\, e_2 \Uparrow \tau_1}$$

array:

$$\frac{\Delta \models e_1 \Downarrow T(Int) \quad \Delta \models c \Downarrow T \quad \Delta \models e_2 \Downarrow T(c)}{\Delta \models array_c(e_1, e_2) \Uparrow T(c\, array)}$$

sub:

$$\frac{\Delta \models e_1 \Uparrow T(c) \quad \Delta \models c \Longrightarrow c'\, array \quad \Delta \models e_2 \Downarrow Int}{\Delta \models sub(e_1, e_2) \Uparrow T(c')}$$

fsub:

$$\frac{\Delta \models e_1 \Downarrow T(BoxFloat\, array) \quad \Delta \models e_2 \Downarrow T(Int)}{\Delta \models fsub(e_1, e_2) \Uparrow Float}$$

2.1.9. *Natural Kind Extraction* $\Delta \models p \rightsquigarrow \kappa$. Assumes that Δ and p are well-formed. Returns the unselfified kind of p.

TS: if Δ ok, $\Delta \vdash p :: \kappa$and $\Delta \models p \rightsquigarrow \kappa'$, then $\Delta \vdash \kappa$.

TS: if Δ ok, $\Delta \vdash p :: \kappa$and $\Delta \models p \rightsquigarrow \kappa'$, then $\Delta \vdash p :: \kappa'$.

Variable:

$$\frac{}{\Delta[\alpha :: \kappa] \models \alpha \rightsquigarrow \kappa}$$

Proj1:

$$\frac{\Delta \models p \rightsquigarrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models p.1 \rightsquigarrow \kappa_1}$$

Proj2:

$$\frac{\Delta \models p \rightsquigarrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models p.2 \rightsquigarrow \{p.1/\alpha\}\kappa_2}$$

App:

$$\frac{\Delta \models p \rightsquigarrow \Pi(\alpha :: \kappa_1).\kappa_2}{\Delta \models p\, c \rightsquigarrow \{c/\alpha\}\kappa_2}$$

2.1.10. *Constructor Weak Head Normal Form* $\Delta \models c \Longrightarrow c'$. Assumes that Δ and c are well-formed. Returns the head normal form of c.

TS: if $\Delta \vdash c :: \kappa$and $\Delta \models c \Longrightarrow c'$, then $\Delta \vdash c' :: \kappa$

Variable:

$$\frac{\Delta \models \alpha \rightsquigarrow S_T(c) \quad \Delta \models c \Longrightarrow c'}{\Delta \models \alpha \Longrightarrow c'}$$

OpaqueVariable:

$$\frac{\Delta \models \alpha \rightsquigarrow \kappa}{\Delta \models \alpha \Longrightarrow \alpha}$$

where κ is not a singleton

BoxFloat:

$$\frac{}{\Delta \models BoxFloat \Longrightarrow BoxFloat}$$

Int:

$$\frac{}{\Delta \models Int \Longrightarrow Int}$$

μ:

$$\frac{}{\Delta \models \mu(a = c_1, b = c_2) \Longrightarrow \mu(a = c_1, b = c_2)}$$

Pair:

$$\frac{}{\Delta \models c_1 \times c_2 \Longrightarrow c_1 \times c_2}$$

Arrow:

$$\frac{}{\Delta \models c_1 \to c_2 \Longrightarrow c_1 \to c_2}$$

Sum:

$$\frac{}{\Delta \models c_1 + c_2 \Longrightarrow c_1 + c_2}$$

Array:

$$\frac{}{\Delta \models c\, array \Longrightarrow c\, array}$$

Lambda:

$$\frac{}{\Delta \models \lambda\alpha :: \kappa.c \Longrightarrow \lambda\alpha :: \kappa.c}$$

App:

$$\frac{\Delta \models c_1 \Longrightarrow \lambda\alpha :: \kappa.c_1 \quad \Delta \models \{c_2/\alpha\}c_1 \Longrightarrow c}{\Delta \models c_1\, c_2 \Longrightarrow c}$$

PathApp:

$$\frac{\begin{array}{l}\Delta \models c_1 \Longrightarrow p \quad \Delta \models p\, c \rightsquigarrow S_T(c') \\ \Delta \models c' \Longrightarrow c''\end{array}}{\Delta \models c_1\, c_2 \Longrightarrow c''}$$

OpaquePathApp:

$$\frac{\Delta \models c_1 \Longrightarrow p \quad \Delta \models p\, c \rightsquigarrow \kappa}{\Delta \models c_1\, c_2 \Longrightarrow p\, c}$$

where κ is not a singleton

Record:

$$\frac{}{\Delta \models < c_1, c_2 > \Longrightarrow < c_1, c_2 >}$$

Proj1:

$$\frac{\Delta \models c \Longrightarrow < c_1, c_2 > \quad \Delta \models c_1 \Longrightarrow c_1'}{\Delta \models c.1 \Longrightarrow c_1'}$$

PathProj1:

$$\frac{\Delta \models c \Longrightarrow p \quad \Delta \models p.1 \rightsquigarrow S_T(c') \quad \Delta \models c' \Longrightarrow c''}{\Delta \models c.1 \Longrightarrow c''}$$

OpaquePathProj1:

$$\frac{\Delta \models c \Longrightarrow p \quad \Delta \models p.1 \rightsquigarrow \kappa}{\Delta \models c.1 \Longrightarrow p.1}$$

where κ is not a singleton

Proj2:

$$\frac{\Delta \models c \Longrightarrow < c_1, c_2 > \quad \Delta \models c_2 \Longrightarrow c_2'}{\Delta \models c.2 \Longrightarrow c_2'}$$

PathProj2:

$$\frac{\Delta \models c \Longrightarrow p \quad \Delta \models p.2 \rightsquigarrow S_T(c') \quad \Delta \models c' \Longrightarrow c''}{\Delta \models c.2 \Longrightarrow c''}$$

OpaquePathProj2:

$$\frac{\Delta \models c \Longrightarrow p \quad \Delta \models p.2 \rightsquigarrow \kappa}{\Delta \models c.2 \Longrightarrow p.2}$$

where κ is not a singleton

Let:

$$\frac{\Delta \models \{c_1/\alpha\}c_2 \Longrightarrow c}{\Delta \models \mathrm{let}\, \alpha = c_1\, \mathrm{in}\, c_2\, \mathrm{end} \Longrightarrow c}$$

2.2. **Termination Proofs.** For the time being, we assume that constructor equivalence at kind type is decidable. For this section, I view the judgments given above as algorithms, as they were intended to be. All of the judgments presented above are algorithmic in the sense that either they are entirely syntax directed, or else at worse, require the "results" of one of their hypotheses to determine a rule which uniquely applies.

2.2.1. *Termination of sub-kinding.* Consider the relation $\prec$ on sub-kinding judgments defined as follows: $(\Delta' \models \kappa_1' \preceq \kappa_2') \prec (\Delta \models \kappa_1 \preceq \kappa_2)$ iff showing $\Delta' \models \kappa_1' \preceq \kappa_2'$ is an immediate sub-goal of showing $\Delta \models \kappa_1 \preceq \kappa_2$. It suffices to show that the $\prec$ relation is well-founded, since if there are no infinite descending chains in the relation, then clearly there are no infinite sequences of rule applications. To show that this is the case, we exhibit a mapping SZ which maps judgments to natural numbers, and show that this map is order preserving.

Definition 1. $SZ(\Delta \models \kappa_1 \preceq \kappa_2) = sz(\kappa_1) + sz(\kappa_2)$, where

$$sz(\kappa) = \begin{cases} 1 & \text{if } \kappa = T \\ 1 & \text{if } \kappa = S_T(c) \\ sz(\kappa_1) + sz(\kappa_2) & \text{if } \kappa = \Sigma(\alpha :: \kappa_1).\kappa_2 \\ sz(\kappa_1) + sz(\kappa_2) & \text{if } \kappa = \Pi(\alpha :: \kappa_1).\kappa_2 \end{cases}$$

Lemma 1. *SZ is a function.*

Proof. Clearly it suffices to show that sz is a function - that is, $\forall \kappa, \exists! n \, s.t. \, sz(\kappa) = n$. This follows by induction over the structure of κ. □

Lemma 2. *SZ is order preserving. That is,*

$$(\Delta' \models \kappa_1' \preceq \kappa_2') \prec (\Delta \models \kappa_1 \preceq \kappa_2) \Rightarrow SZ(\Delta' \models \kappa_1' \preceq \kappa_2') < SZ(\Delta \models \kappa_1 \preceq \kappa_2)$$

Proof. We proceed by cases on the subkinding judgements that define$\prec$.

(1) $\Delta \models T \preceq T$. Vacuously true - the rule has no premises, and hence has nothing smaller than it.

(2) $\Delta \models S_T(c) \preceq T$. Vacuously true - the rule has no premises, and hence has nothing smaller than it.

(3) $\Delta \models S_T(c) \preceq S_T(d)$. This rule has no sub-kinding premises, and hence has nothing smaller than it. The only subgoal is an equivalence judgment, which we assume terminates.

(4) $\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'$. The subkinding rule for the Π judgment defines two judgments as smaller:

(a) By definition, $(\Delta \models \kappa_1' \preceq \kappa_1) \prec (\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2')$. But note that

$$\begin{aligned} SZ(\Delta \models \kappa_1' \preceq \kappa_1) &= sz(\kappa_1') + sz(\kappa_1) \\ &< sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ &= SZ(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2') \end{aligned}$$

(b) By definition, $(\Delta[\alpha :: \kappa_1'] \models \kappa_2 \preceq \kappa_2') \prec (\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2')$. But note that

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa_1'] \models \kappa_2 \preceq \kappa_2') &= sz(\kappa_2') + sz(\kappa_2) \\ &< sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ &= SZ(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2') \end{aligned}$$

(5) $\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'$. The subkinding rule for the Σ judgment defines two judgements as smaller:

(a) By definition, $\Delta \models \kappa_1 \preceq \kappa_1' \prec \Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'$. But note that

$$\begin{aligned} SZ(\Delta \models \kappa_1 \preceq \kappa_1') &= sz(\kappa_1') + sz(\kappa_1) \\ &< sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ &= SZ(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2') \end{aligned}$$

(b) By definition, $\Delta[\alpha :: \kappa_1] \models \kappa_2 \preceq \kappa_2' \prec \Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'$. But note that

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa_1] \models \kappa_2 \preceq \kappa_2') &= sz(\kappa_2') + sz(\kappa_2) \\ &< sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ &= SZ(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2') \end{aligned}$$

□

Theorem 1. *The algorithm for checking subkinding always terminates. That is, the algorithmic rules for subkinding do not permit any infinite sequences of rule applications.*

Proof. By the previous lemmas, every rule uses only premises that are strictly smaller than the conclusion according to a well-founded ordering. Therefore, there can be no infinite sequence of rule applications, since such a sequence would correspond to an infinite descending chain in the well-founded ordering. □

2.2.2. *Termination of the well-formed kind, kind analysis, and kind synthesis algorithms.* We start by defining measure functions which map judgments to pairs of natural numbers ordered lexicographically below. These functions are defined in terms of inductive defined functions $sz_\kappa()$ and $sz_c()$, which act as measures on kinds and constructors, respectively.

Definition 2.

$$sz_\kappa(\kappa) = \begin{cases} 1 & \text{if } \kappa = T \\ sz_c(c) + 1 & \text{if } \kappa = S_T(c) \\ sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2) & \text{if } \kappa = \Sigma(\alpha :: \kappa_1).\kappa_2 \\ sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2) & \text{if } \kappa = \Pi(\alpha :: \kappa_1).\kappa_2 \end{cases}$$

$$sz_c(c) = \begin{cases} 1 & \text{if } c = \alpha, Int, BoxFloat \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \mu(a = c_1, b = c_2) \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1 \times c_2, c_1 \to c_2, c_1 + c_2 \\ sz_c(c') + 1 & \text{if } c = c'\ array \\ sz_c(c') + sz_\kappa(\kappa) & \text{if } c = \lambda\alpha :: \kappa.c' \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1\, c_2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c =< c_1, c_2 > \\ sz_c(c') + 1 & \text{if } c = c'.1, c'.2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \text{let } \alpha = c_1 \text{ in } c_2 \text{ end} \end{cases}$$

Definition 3. Well-formed kind measure. $SZ_\kappa(\Delta \models \kappa) = (sz_\kappa(\kappa), 0)$

Definition 4. Type analysis measure. $SZ_\Downarrow(\Delta \models c \Downarrow \kappa) = (sz_c(c), 1)$

Definition 5. Type synthesis measure. $SZ_\Uparrow(\Delta \models c \Uparrow \kappa) = (sz_c(c), 0)$

Lemma 3. *$SZ_\kappa()$, $SZ_\Uparrow()$, and $SZ_\Downarrow()$ are well-defined functions.*

Proof. It clearly suffices to show that $sz_c()$, and $sz_\kappa()$ are well-defined. This follows by induction over the structure of κ and c. □

Lemma 4. *Termination of selfification.*

Proof. Follows immediately by induction over the structure of κ. □

Lemma 5. *For each rule in the well-formed kind, kind synthesis and kind analysis judgments, the measure of each premise is smaller than the measure of the conclusion.*

Proof. The proof proceeds by cases over the rules, demonstrating that each premise has a measure which is strictly smaller than the measure of the conclusion of the rule. We ignore premises that correspond to judgements which are independently known or assumed to terminate, such as subkinding and constructor equivalence. Technically, this may be viewed as using the constant measure that always returns zero for these judgements.

- Well Formed Kind $\Delta \models \kappa$
  Type: No premises.
  Singleton:

$$\begin{aligned} SZ_{\Downarrow}(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= SZ_\kappa(\Delta \models S_T(c)) \end{aligned}$$

  Pi:
  (1)

$$\begin{aligned} SZ_\kappa(\Delta \models \kappa_1) &= (sz_\kappa(\kappa_1), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ_\kappa(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

  (2)

$$\begin{aligned} SZ_\kappa(\Delta[\alpha :: \kappa_1] \models \kappa_2) &= (sz_\kappa(\kappa_2), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ_\kappa(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

  Sigma:
  (1)

$$\begin{aligned} SZ_\kappa(\Delta \models \kappa_1) &= (sz_\kappa(\kappa_1), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ_\kappa(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

  (2)

$$\begin{aligned} SZ_\kappa(\Delta[\alpha :: \kappa_1] \models \kappa_2) &= (sz_\kappa(\kappa_2), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ_\kappa(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

- Kind Analysis $\Delta \models c \Downarrow \kappa$. Here is where the second component of the size metric is used.

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c \Uparrow \kappa') &= (sz_c(c), 0) \\ &< (sz_c(c), 1) \\ &= SZ_{\Downarrow}(\Delta \models c \Downarrow \kappa) \end{aligned}$$

- Kind Synthesis $\Delta \models c \Uparrow \kappa$
  Variable: By lemma **??**
  BoxFloat: No premises
  Int: No premises
  μ**:**
  (1)

$$\begin{aligned} SZ_{\Downarrow}(\Delta[a :: T, b :: T] \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(a = c_1, b = c_2)), 0) \\ &= SZ_{\Uparrow}(\Delta \models \mu(a = c_1, b = c_2) \Uparrow \kappa) \end{aligned}$$

  where $\kappa = S_T(\mu(a = c_1, b = c_2).1) \times S_T(\mu(a = c_1, b = c_2).2)$.

  (2) Similar
  Pair:
  (1)

$$\begin{aligned} SZ_{\Downarrow}(\Delta \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1 \times c_2), 0) \\ &= SZ_{\Uparrow}(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1 \times c_2)) \end{aligned}$$

  (2) Similarly for the second premise.
  Arrow: As with the Pair case.
  Sum: As with the Pair case.
  Array:

$$\begin{aligned} SZ_{\Downarrow}(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= (sz_c(c\, array), 0) \\ &= SZ_{\Uparrow}(\Delta \models c\, array \Uparrow S_T(c\, array)) \end{aligned}$$

  Lambda:
  (1)

$$\begin{aligned} SZ_{\kappa}(\Delta \models \kappa) &= (sz_{\kappa}(\kappa), 0) \\ &< (sz_{\kappa}(\kappa) + sz_c(c), 0) \\ &= (sz_c(\lambda\alpha :: \kappa.c), 0) \\ &= SZ_{\Uparrow}(\Delta \models \lambda\alpha :: \kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

(2)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta[\alpha :: \kappa] \models c \Uparrow \kappa') & = & (sz_c(c), 0) \\ & < & (sz_\kappa(\kappa) + sz_c(c), 0) \\ & = & (sz_c(\lambda\alpha :: \kappa.c), 0) \\ & = & SZ_{\Uparrow}(\Delta \models \lambda\alpha :: \kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{array}$$

App:

(1)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2) & = & (sz_c(c_1), 0) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (sz_c(c_1\, c_2), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2) \end{array}$$

(2)

$$\begin{array}{rcl} SZ_{\Downarrow}(\Delta \models c_2 \Downarrow \kappa_1) & = & (sz_c(c_2), 1) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (sz_c(c_1\, c_2), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2) \end{array}$$

Record:

(1)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \kappa_1) & = & (sz_c(c_1), 0) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (, sz_c(< c_1, c_2 >), 0) \\ & = & SZ_{\Uparrow}(\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2) \end{array}$$

(2)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c_2 \Uparrow \kappa_2) & = & (sz_c(c_2), 0) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (sz_c(< c_1, c_2 >), 0) \\ & = & SZ_{\Uparrow}(\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2) \end{array}$$

Proj1:

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2) & = & (sz_c(c), 0) \\ & < & (sz_c(c) + 1, 0) \\ & = & (sz_c(c.1), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c.1 \Uparrow \kappa_1) \end{array}$$

Proj2: As with Proj1

Let:

(1)

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \kappa_1) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\text{let } \alpha = c_1 \text{ in } c_2 \text{ end}), 0) \\ &= SZ_{\Uparrow}(\Delta \models \text{let } \alpha = c_1 \text{ in } c_2 \text{ end} \Uparrow \{c_1/\alpha\}\kappa_2) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\Uparrow}(\Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2) &= (sz_c(c_2), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\text{let } \alpha = c_1 \text{ in } c_2 \text{ end}), 0) \\ &= SZ_{\Uparrow}(\Delta \models \text{let } \alpha = c_1 \text{ in } c_2 \text{ end} \Uparrow \{c_1/\alpha\}\kappa_2) \end{aligned}$$

□

Theorem 2. *The rules for kind synthesis, kind analysis, and kind well-formedness do not permit an infinite sequence of rule applications.*

Proof. By lemma **??**, any infinite sequence of rule applications corresponds to an infinite descending chain of pairs of natural numbers ordered lexicographically, which contradicts the well-foundedness of $(N \times N, <)$. □

3. NIL (Extended MIL)

3.1. **Syntax.** This is a slight simplification of the syntactic forms of the NIL. In particular, to be completely faithful to the implementation, there should be an intermediate syntactic form, which at the kind level corresponds to the greek syntax, except in that the contents of singletons may be constructors drawn from the roman syntax. In such a system, I believe that the constructor standardization rules do not play a part in the algorithm - only in the statement of the theorems.

$$
\begin{array}{rcl}
k & ::= & S(c) \mid T \mid S_T(c) \mid \Sigma(\alpha :: k).k \mid \Pi(\alpha :: k).k \\
\\
c & ::= & \alpha \mid Int \mid BoxFloat \mid \mu(a = c, b = c) \mid c \times c \mid c \to c \mid c + c \mid c\, array \\
& & \mid \lambda\alpha :: k.c \mid c\, c \mid < c, c > \mid c.1 \mid c.2 \mid \mathrm{let}\, \alpha = c\, \mathrm{in}\, c\, \mathrm{end} \\
\\
t & ::= & \mathrm{Typeof}(e) \mid T(c) \mid (\alpha_1 :: k_1 \ldots \alpha_n :: k_n, x : t) \to t \mid Float \mid t \times t \\
\\
p & ::= & \alpha \mid p.1 \mid p.2 \mid p\, c \\
\\
e & ::= & x \mid \mathrm{let}\, x = e\, \mathrm{in}\, e\, \mathrm{end} \mid \mathrm{let}\, \alpha = c\, \mathrm{in}\, e\, \mathrm{end} \\
& & \mid \mathrm{rec}\, f = \lambda(\alpha_1 :: k_1 \ldots \alpha_n :: \kappa_n, x : t) : t.e \\
& & \mid e[c_1 \ldots c_n]e \mid < e, e > \mid e.1 \mid e.2 \mid n \mid r \mid boxfloat(e) \mid unboxfloat(e) \\
& & \mid inl_{c,c}e \mid inr_{c,c}e \mid case\, e\, of\, \{inl(x : c) \Rightarrow e, inr(x : c) \Rightarrow e\} \\
& & \mid \mathrm{ifBoxFloat}\, c\, \mathrm{then}\, e\, \mathrm{else}\, e \mid array_c(e, e) \mid sub(e, e) \mid fsub(e, e) \\
\\
\Delta & ::= & \bullet \mid \Delta[x : \tau] \mid \Delta[\alpha :: \kappa]
\end{array}
$$

Note that contexts are restricted to the core syntactic forms.

3.2. **Judgments.**

3.2.1. *Kind Standardization* $\Delta \models k \backslash \kappa$. Asumes that Δ and k are well-formed (as defined below), and constructs a κ such that κ is well-formed as defined above.

TS: if Δ ok, $\Delta \models k$, and $\Delta \models k \backslash \kappa$ then $\Delta \vdash \kappa$

Type:

$$\frac{}{\Delta \models T \backslash T}$$

Singleton Type:

$$\frac{\Delta \models c \backslash c'}{\Delta \models S_T(c) \backslash S_T(c')}$$

Singleton Any:

$$\frac{\Delta \models c \Uparrow \kappa}{\Delta \models S(c) \backslash \kappa}$$

Pi:

$$\frac{\Delta \models k_1 \backslash \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models k_2 \backslash \kappa_2}{\Delta \models \Pi(\alpha :: k_1).k_2 \backslash \Pi(\alpha :: \kappa_1).\kappa_2}$$

Sigma:

$$\frac{\Delta \models k_1 \backslash \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models k_2 \backslash \kappa_2}{\Delta \models \Sigma(\alpha :: k_1).k_2 \backslash \Sigma(\alpha :: \kappa_1).\kappa_2}$$

3.2.2. *Constructor standardization* $\Delta \models c \backslash c'$. Assumes that Δ and c are well-formed in the sense defined below. Constructs an "equivalent" c' that is well-formed in the declarative sense.

TS: if $\Delta\,\mathrm{ok}$, $\Delta \models c \Uparrow \kappa$, and $\Delta \models c \backslash c'$ then $\Delta \vdash c :: \kappa$

All cases proceed compositionally over the structure of the constructors except for the following cases:

Lambda:

$$\frac{\Delta \models k \backslash \kappa \quad \Delta[\alpha :: \kappa] \models c \backslash c'}{\Delta \models \lambda\alpha :: k.c \backslash \lambda\alpha :: \kappa.c'}$$

Let: (Note that the kind for c_1 could also be synthesized from c_1', but the proof of termination then gets harder)

$$\frac{\begin{array}{ll}\Delta \models c_1 \backslash c_1' & \Delta \models c_1 \Uparrow \kappa \\ \Delta[\alpha :: \kappa] \models c_2 \backslash c_2' & \end{array}}{\Delta \models \mathrm{let}\,\alpha = c_1\,\mathrm{in}\,c_2\,\mathrm{end} \backslash \mathrm{let}\,\alpha = c_1'\,\mathrm{in}\,c_2'\,\mathrm{end}}$$

3.2.3. *Type standardization* $\Delta \models t \backslash \tau$. Assumes that Δ and t are well-formed in the sense defined below. Constructs an "equivalent" τ that is well-formed in the declarative sense.

TS: if $\Delta\,\mathrm{ok}$, $\Delta \models t$, and $\Delta \models t \backslash \tau$, then $\Delta \vdash \tau$

Typeof:

$$\frac{\Delta \models e \Uparrow \tau}{\Delta \models \mathrm{Typeof}(e) \backslash \tau}$$

Constructor:

$$\frac{\Delta \models c \backslash c'}{\Delta \models T(c) \backslash T(c')}$$

Arrow:

$$\frac{\begin{array}{ll}\Delta[\alpha_i :: \kappa_i] \models k_{i+1} \backslash \kappa_{i+1} & \Delta[\alpha_n :: \kappa_n] \models t_1 \backslash \tau_1 \\ \Delta[\alpha_n :: \kappa_n][x : \tau_1] \models t_2 \backslash \tau_2 & \end{array}}{\Delta \models (\alpha_1 :: k_1 \ldots \alpha_n :: k_n, x : t_1) \to t_2 \backslash (\alpha_1 :: \kappa_1 \ldots \alpha_n :: \kappa_n, \tau_1) \to \tau_2}$$

Float:

$$\frac{}{\Delta \models \mathit{Float} \backslash \mathit{Float}}$$

Pair:

$$\frac{\Delta \models t_1 \backslash \tau_1 \quad \Delta \models t_2 \backslash \tau_2}{\Delta \models t_1 \times t_2 \backslash \tau_1 \times \tau_2}$$

3.2.4. *Expression standardization* $\Delta \models e \backslash e'$. This judgement is not necessary for the algorithm, but is necessary for stating properties that should hold with respect to the declarative system.

3.2.5. *Well Formed Kind* $\Delta \models k$. Assumes that Δ is well-formed, asserts that k is well-formed.

TS: if Δ ok, $\Delta \models k$ and $\Delta \models k \backslash \kappa$, then $\Delta \vdash \kappa$.

TS: if $\Delta \vdash \kappa$ then $\Delta \models \kappa$. (Technically, there is an identity coercion here mapping the κ to a k. Or alternatively, just view the NIL as containing the MIL as well)

Type: As before

Singleton Type: As before

Singleton Any:

$$\frac{\Delta \models c \Downarrow T}{\Delta \models S(c)}$$

Pi:

$$\frac{\Delta \models k_1 \qquad \Delta \models k_1 \backslash \kappa_1 \qquad \Delta[\alpha :: \kappa_1] \models k_2}{\Delta \models \Pi(\alpha :: k_1).k_2}$$

Sigma:

$$\frac{\Delta \models k_1 \qquad \Delta \models k_1 \backslash \kappa_1 \qquad \Delta[\alpha :: \kappa_1] \models k_2}{\Delta \models \Sigma(\alpha :: k_1).k_2}$$

3.2.6. *Sub-Kinding* $\Delta \models \kappa_1 \preceq \kappa_2$. We do not choose to define subkinding for the extended NIL - all queries will be restricted to core syntax.

3.2.7. *Kind Analysis* $\Delta \models c \Downarrow \kappa$. Note that we restrict this judgement to core kinds. Assume Δ and κ are well formed. Check that c is well formed and can be given kind k.

TS: if $\Delta \vdash c :: \kappa$ then $\Delta \models c \Downarrow \kappa$.

TS: if Δ ok, $\Delta \vdash \kappa$, $\Delta \models c \Downarrow \kappa$, then $\Delta \vdash c :: \kappa$

$$\frac{\Delta \models c \Uparrow \kappa' \quad \Delta \models \kappa' \preceq \kappa}{\Delta \models c \Downarrow \kappa}$$

3.2.8. *Kind Synthesis* $\Delta \models c \Uparrow k$. Assumes that Δ is well-formed. Check that c is well-kinded, and construct κ s.t. $\Delta \models \kappa$ and c has kind κ.

TS: if $\Delta \vdash c :: \kappa$ then $\Delta \models c \Downarrow \kappa$

TS: if Δ ok, $\Delta \models c \Uparrow \kappa$, and $\Delta \models c \backslash c'$ then $\Delta \vdash c' :: \kappa$.

Variable:

$$\frac{\alpha :: \kappa \doteq \kappa'}{\Delta[\alpha :: \kappa] \models \alpha \Uparrow \kappa'}$$

BoxFloat:

$$\frac{}{\Delta \models BoxFloat \Uparrow S_T(BoxFloat)}$$

Int:

$$\frac{}{\Delta \models Int \Uparrow S_T(Int)}$$

μ:

$$\frac{\begin{array}{cc}\Delta[a::T,b::T] \models c_1 \Downarrow T & \Delta[a::T,b::T] \models c_2 \Downarrow T \\ \Delta[a::T,b::T] \models c_1 \backslash c_1' & \Delta[a::T,b::T] \models c_2 \backslash c_2'\end{array}}{\Delta \models \mu(a = c_1, b = c_2) \Uparrow S_T(\mu(a = c_1', b = c_2').1) \times S_T(\mu(a = c_1', b = c_2').2)} \quad a, b \notin dom(\Delta)$$

Pair:

$$\frac{\begin{array}{cc}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')}$$

Arrow:

$$\frac{\begin{array}{cc}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 \to c_2 \Uparrow S_T(c_1' \to c_2')}$$

Sum:

$$\frac{\begin{array}{cc}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 + c_2 \Uparrow S_T(c_1' + c_2')}$$

Array:

$$\frac{\Delta \models c \Downarrow T \quad \Delta \models c \backslash c'}{\Delta \models c\, array \Uparrow S_T(c'\, array)}$$

Lambda:

$$\frac{\begin{array}{cc}\Delta \models k & \Delta \models k \backslash \kappa \\ \Delta[\alpha :: \kappa] \models c \Uparrow \kappa' & \end{array}}{\Delta \models \lambda \alpha :: k.c \Uparrow \Pi(\alpha :: \kappa).\kappa'} \quad \alpha \notin dom(\Delta)$$

App:

$$\frac{\begin{array}{cc}\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2 & \Delta \models c_2 \Downarrow \kappa_1 \\ \Delta \models c_2 \backslash c_2' & \end{array}}{\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2}$$

Record:

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta \models c_2 \Uparrow \kappa_2}{\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2}$$

Proj1:

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.1 \Uparrow \kappa_1}$$

Proj2:

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \models c \backslash c'}{\Delta \models c.2 \Uparrow \{c'.1/\alpha\}\kappa_2}$$

Let:

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2 \quad \Delta \models c_1 \backslash c_1'}{\Delta \models \text{let}\, \alpha = c_1 \,\text{in}\, c_2 \,\text{end} \Uparrow \{c_1'/\alpha\}\kappa_2} \quad \alpha \notin dom(\Delta)$$

There should be a lemma showing that for let binding at least, the substitution is not necessary since the kinds are principle, and since selfification chooses to preserve the contents of singletons. Maybe for record kinds as well?

3.2.9. *Well-formed Type* $\Delta \models t$. Assume Δ is well-formed. Check that t is well-formed.
TS: if Δ ok, $\Delta \models t$, and $\Delta \models t \backslash \tau$ then $\Delta \vdash \tau$.
TS: if $\Delta \vdash \tau$ then $\Delta \models \tau'$ such that $\Delta \vdash \tau \equiv \tau'$

Constructor:

$$\frac{\Delta \models c \Downarrow T}{\Delta \models T(c)}$$

ArrowType:

$$\frac{\begin{array}{ll}\Delta[\alpha_i :: \kappa_i] \models k_{i+1} & \Delta[\alpha_i :: \kappa_i] \models k_{i+1} \backslash \kappa_{i+1} \\ \Delta[\alpha_n :: \kappa_n] \models t_1 & \Delta[\alpha_n :: \kappa_n] \models t_1 \backslash \tau_1 \\ \Delta[\alpha_i :: \kappa_i][x : \tau_1] \models t_2 & \end{array}}{\Delta \models (\alpha_n :: k_n, x : t_1) \to t_2} \quad \alpha_{i+1} \notin dom(\Delta[\alpha_i :: \kappa_i])$$

Float:

$$\frac{}{\Delta \models Float}$$

PairType:

$$\frac{\Delta \models t_1 \quad \Delta \models t_2}{\Delta \models t_1 \times t_2}$$

3.2.10. *Type Analysis* $\Delta \models e \Downarrow \tau$. Note that we restrict this to core types. Assume Δ and t are well-formed. Check that e is well-typed, and has type τ.

$$\frac{\Delta \models e \Uparrow \tau' \quad \Delta \models \tau' \equiv \tau}{\Delta \models e \Downarrow \tau}$$

3.2.11. *Type Synthesis* $\Delta \models e \Uparrow \tau$. Assume Δ is well-formed. Check that e is well-formed and construct its type τ, such that $\Delta \models \tau$
TS: if Δ ok and $\Delta \models e \Uparrow \tau$, and $\Delta \models e \backslash e'$then $\Delta \vdash e' : \tau$
TS if $\Delta \vdash e : \tau$, then $\Delta \models e \Uparrow \tau'$ such that $\Delta \vdash \tau \equiv \tau'$

variable:

$$\frac{}{\Delta[x : \tau] \models x \Uparrow \tau}$$

lete:

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta[x : \tau_1] \models e_2 \Uparrow \tau_2}{\Delta \models \text{let}\, x = e_1 \,\text{in}\, e_2 \,\text{end} \Uparrow \tau_2} \quad x \notin dom(\Delta)$$

letc:

$$\frac{\Delta \models c \Uparrow \kappa \qquad \Delta \models c \backslash c' \quad \Delta[\alpha :: \kappa] \models e \Uparrow \tau}{\Delta \models \text{let}\, \alpha = c \,\text{in}\, e \,\text{end} \Uparrow \{c'/\alpha\}\tau} \quad \alpha \notin dom(\Delta)$$

rec:

$$\frac{\begin{array}{ll}\Delta[\alpha_i :: \kappa_i] \models k_{i+1} & \Delta[\alpha :: \kappa_i] \models k_{i+1} \backslash \kappa_{i+1} \\ \Delta[\alpha_n :: \kappa_n] \models t_1 & \Delta[\alpha_n :: \kappa_n] \models t_1 \backslash \tau_1 \\ \Delta[\alpha_n :: \kappa_n][x : \tau_1] \models t_2 & \Delta[\alpha_n :: \kappa_n][x : \tau_1] \models t_2 \backslash \tau_2 \\ \Delta[f : (\alpha_n :: \kappa_n, \tau_1) \to \tau_2][\alpha_n :: \kappa_n][x : \tau_1] \models e \Downarrow \tau_2 & \end{array}}{\Delta \models \text{rec}\, f = \lambda(\alpha_n :: k_n, x : t_1) : t_2.e \Uparrow (\alpha_n :: \kappa_n, \tau_1) \to \tau_2}$$

where $f, x \notin dom(\Delta), \alpha_{i+1} \notin dom(\Delta[\alpha_i :: \kappa_i])$.

app:

$$\frac{\begin{array}{ll}\Delta \models e_1 \Uparrow (\alpha_1 :: \kappa_1 \ldots \alpha_n :: \kappa_n, \tau_1) \to \tau_2 & \Delta \models c_{i+1} \Downarrow \{c_i'/\alpha_i\}\kappa_{i+1} \\ \Delta \models c_i \backslash c_i' & \Delta \models e_2 \Downarrow \{c_n'/\alpha_n\}\tau_1 \end{array}}{\Delta \models e_1[c_1 \ldots c_n]e_2 \Uparrow \{c_n'/\alpha_n\}\tau_2}$$

mono_con_app:

$$\frac{\Delta \models e_1 \Uparrow T(c_e) \quad \Delta \models c_e \Longrightarrow c_1 \to c_2 \quad \Delta \models e_2 \Downarrow T(c_1)}{\Delta \models e_1[]e_2 \Uparrow T(c_2)}$$

pair:

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta \models e_2 \Uparrow \tau_2}{\Delta \models < e_1, e_2 > \Uparrow \tau_1 \times \tau_2}$$

type_proj1:

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.1 \Uparrow \tau_1}$$

con_proj1:

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \Longrightarrow c_1 \times c_2}{\Delta \models e.1 \Uparrow T(c_1)}$$

type_proj2:

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.2 \Uparrow \tau_2}$$

con_proj2:

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \Longrightarrow c_1 \times c_2}{\Delta \models e.2 \Uparrow T(c_2)}$$

float:

$$\frac{}{\Delta \models r \Uparrow Float}$$

int:

$$\frac{}{\Delta \models n \Uparrow T(Int)}$$

box:

$$\frac{\Delta \models e \Downarrow Float}{\Delta \models boxfloat(e) \Uparrow T(BoxFloat)}$$

unbox:

$$\frac{\Delta \models e \Downarrow T(BoxFloat)}{\Delta \models unboxfloat(e) \Uparrow Float}$$

sumswitch:

$$\frac{\begin{array}{ll}\Delta \models c_1 \Downarrow T & \Delta \models c_1 \backslash c_1' \\ \Delta \models c_2 \Downarrow T & \Delta \models c_2 \backslash c_2' \\ \Delta[x_1 : T(c_1')] \models e_1 \Uparrow \tau_1 & \Delta[x_2 : T(c_2')] \models e_2 \Uparrow \tau_2 \\ \Delta \models e \Downarrow T(c_1' + c_2') & \Delta \models \tau_1 \equiv \tau_2\end{array}}{\Delta \models case\, e\, of\, \{inl(x_1 : c_1) \Rightarrow e_1, inr(x_2 : c_2) \Rightarrow e_2\} \Uparrow \tau_1}$$

inl:

$$\frac{\begin{array}{ll}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2' \\ \Delta \models e \Downarrow T(c_1') & \end{array}}{\Delta \models inl_{c_1,c_2} e \Uparrow T(c_1' + c_2')}$$

inr:

$$\frac{\begin{array}{ll}\Delta \models c \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2' \\ \Delta \models e \Downarrow T(c_2') & \end{array}}{\Delta \models inr_{c_1,c_2} e \Uparrow T(c_1' + c_2')}$$

ifBoxFloat:

$$\frac{\begin{array}{ll}\Delta \models c \Downarrow T & \Delta \models e_1 \Uparrow \tau_1 \\ \Delta \models e_2 \Uparrow \tau_2 & \Delta \models \tau_1 \equiv \tau_2\end{array}}{\Delta \models \text{ifBoxFloat}\, c\, \text{then}\, e_1\, \text{else}\, e_2 \Uparrow \tau_1}$$

array:

$$\frac{\begin{array}{ll}\Delta \models c \Downarrow T & \Delta \models c \backslash c' \\ \Delta \models e_1 \Downarrow T(Int) & \Delta \models e_2 \Downarrow T(c')\end{array}}{\Delta \models array_c(e_1, e_2) \Uparrow T(c'\ array)}$$

sub:

$$\frac{\Delta \models e_1 \Uparrow T(c) \quad \Delta \models c \Longrightarrow c'\ array \quad \Delta \models e_2 \Downarrow Int}{\Delta \models sub(e_1, e_2) \Uparrow T(c')}$$

fsub:

$$\frac{\Delta \models e_1 \Downarrow T(BoxFloat\ array) \quad \Delta \models e_2 \Downarrow T(Int)}{\Delta \models fsub(e_1, e_2) \Uparrow Float}$$

3.3. **Termination Proofs.** We assume that constructor equivalence at kind type *for the core NIL* is decidable.

3.3.1. *Termination of the kind standardization, constructor standardization, well-formed kind, kind analysis, and kind synthesis algorithms.* We start by defining measure functions which map judgments to pairs of natural numbers ordered lexicographically below. These functions are defined in terms of the inductively defined functions $sz_k()$ and $sz_c()$.

Definition 6.

$$sz_k(k) = \begin{cases} 1 & \text{if } k = T \\ sz_c(c) + 1 & \text{if } k = S_T(c) \\ sz_c(c) + 1 & if\ k = S(c) \\ sz_k(k_1) + sz_k(k_2) & \text{if } k = \Sigma(\alpha :: k_1).k_2 \\ sz_k(k_1) + sz_k(k_2) & \text{if } k = \Pi(\alpha :: k_1).k_2 \end{cases}$$

$$sz_c(c) = \begin{cases} 1 & \text{if } c = \alpha, Int, BoxFloat \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \mu(a = c_1, b = c_2) \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1 \times c_2, c_1 \to c_2, c_1 + c_2 \\ sz_c(c') + 1 & \text{if } c = c'\ array \\ sz_c(c') + sz_k(k) & \text{if } c = \lambda\alpha :: k.c' \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1\, c_2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c =< c_1, c_2 > \\ sz_c(c') + 1 & \text{if } c = c'.1, c'.2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \text{let } \alpha = c_1 \text{ in } c_2 \text{ end} \end{cases}$$

Definition 7. Kind standardization. $SZ_{\backslash k}(\Delta \models k \backslash \kappa) = (sz_k(k), 0)$

Definition 8. Constructor standardization $SZ_{\backslash c}(\Delta \models c \backslash c') = (sz_c(c), 0)$

Definition 9. Well-formed kind measure. $SZ_k(\Delta \models k) = (sz_k(k), 0)$

Definition 10. Type analysis measure. $SZ_{\Downarrow}(\Delta \models c \Downarrow \kappa) = (sz_c(c), 1)$

Definition 11. Type synthesis measure. $SZ_{\Uparrow}(\Delta \models c \Uparrow \kappa) = (sz_c(c), 0)$

Lemma 6. *$SZ_k()$, $SZ_{\Uparrow}()$, $SZ_{\Downarrow}()$, $SZ_{\backslash k}()$, and $SZ_{\backslash c}()$ are well-defined functions.*

Proof. It clearly suffices to show that $sz_c()$, and $sz_k()$ are well-defined. This follows by induction over the structure of k and c. □

Lemma 7. *For each rule in the kind standardization, constructor standardization, well-formed kind, kind synthesis and kind analysis judgments, the measure of each premise is smaller than the measure of the conclusion.*

Proof. The proof proceeds by cases over the rules, demonstrating that each premise has a measure which is strictly smaller than the measure of the conclusion of the rule. We ignore premises that correspond to judgements which are independently known or assumed to terminate, such as subkinding and constructor equivalence. Rules which remain unchanged from the core syntax proof are omitted.

- Kind standardization $\Delta \models k \backslash \kappa$
  Type: No premises
  Singleton_Type:

$$\begin{aligned} SZ_{\backslash c}(\Delta \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= SZ_{\backslash k}(\Delta \models S_T(c) \backslash S_T(c')) \end{aligned}$$

Singleton_Any:

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c \Uparrow \kappa) &= (sz_c(c), 0) \\ &< (sz_c(c)+1, 0) \\ &= SZ_{\backslash k}(\Delta \models S(c) \backslash \kappa) \end{aligned}$$

Pi:

(1)

$$\begin{aligned} SZ_{\backslash k}(\Delta \models k_1 \backslash \kappa_1) &= (sz_k(k_1), 0) \\ &< (sz_k(\Pi(\alpha :: k_1).k_2), 0) \\ &= SZ_{\backslash k}(\Delta \models \Pi(\alpha :: k_1).k_2 \backslash \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash k}(\Delta[\alpha :: \kappa_1] \models k_2 \backslash \kappa_2) &= (sz_k(k_2), 0) \\ &< (sz_k(\Pi(\alpha :: k_1).k_2), 0) \\ &= SZ_{\backslash k}(\Delta \models \Pi(\alpha :: k_1).k_2 \backslash \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

Sigma: As with the Pi case.

- Constructor standardization (All cases except those below are just decomposition of the constructor)

Lambda:

(1)

$$\begin{aligned} SZ_{\backslash k}(\Delta \models k \backslash \kappa) &= (sz_k(k), 0) \\ &< (sz_c(\lambda\alpha :: k.c), 0) \\ &= SZ_{\backslash c}(\Delta \models \lambda\alpha :: k.c \backslash \lambda\alpha :: \kappa.c') \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash c}(\Delta[\alpha :: \kappa] \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(\lambda\alpha :: k.c), 0) \\ &= SZ_{\backslash c}(\Delta \models \lambda\alpha :: k.c \backslash \lambda\alpha :: \kappa.c') \end{aligned}$$

Let: Note that the size of the conclusion is $SZ_{\backslash c}(\Delta \models \mathrm{let}\,\alpha = c_1\,\mathrm{in}\,c_2\,\mathrm{end} \backslash \mathrm{let}\,\alpha = c_1'\,\mathrm{in}\,c_2'\,\mathrm{end}) = (sz_c(c_1) + sz_c(c_2), 0)$

(1)

$$\begin{aligned} SZ_{\backslash c}(\Delta \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \kappa) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \end{aligned}$$

(3)

$$\begin{aligned} SZ_{\backslash c}(\Delta[\alpha :: \kappa] \models c_2 \backslash c_2') &= (sz_c(c_2), 0) \\ &< (sz_c(c_2) + sz_c(c_1), 0) \end{aligned}$$

- Well Formed Kind $\Delta \models k$

Type, Singleton_Type: As before
Singleton_Any:

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c \Uparrow \kappa) &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= SZ_k(\Delta \models S(c)) \end{aligned}$$

Pi:

(1)

$$\begin{aligned} SZ_k(\Delta \models k_1) &= (sz_k(k_1), 0) \\ &< (sz_k(k_1) + sz_k(k_2), 0) \\ &= SZ_k(\Delta \models \Pi(\alpha :: k_1).k_2) \end{aligned}$$

(2)

$$\begin{aligned} SZ_k(\Delta[\alpha :: \kappa_1] \models k_2) &= (sz_k(k_2), 0) \\ &< (sz_k(k_1) + sz_k(k_2), 0) \\ &= SZ_k(\Delta \models \Pi(\alpha :: k_1).k_2) \end{aligned}$$

Sigma: As with the Pi case.

- Kind Analysis remains unchanged.
- Kind Synthesis $\Delta \models c \Uparrow \kappa$

Variable: By lemma **??**. Note that kinds in the context are restricted to the core syntactic forms.
BoxFloat: No premises
Int: No premises
μ: Let $\kappa = S_T(\mu(a = c_1, b = c_2).1) \times S_T(\mu(a = c_1', b = c_2').2)$
(1)

$$\begin{aligned} SZ_{\Downarrow}(\Delta[a :: T, b :: T] \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(a = c_1, b = c_2)), 0) \\ &= SZ_{\Uparrow}(\Delta \models \mu(a = c_1, b = c_2) \Uparrow \kappa) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash c}(\Delta[a :: T, b :: T] \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(a = c_1, b = c_2)), 0) \end{aligned}$$

(3) The cases for c_2 are exactly the same.
Pair:

(1)

$$\begin{aligned} SZ_{\Downarrow}(\Delta \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1 \times c_2), 0) \\ &= SZ_{\Uparrow}(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash_c}(\Delta \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= SZ_{\Uparrow}(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')) \end{aligned}$$

(3) Similarly for the c_2 premises.

Arrow: As with the Pair case.

Sum: As with the Pair case.

Array:

(1)

$$\begin{aligned} SZ_{\Downarrow}(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= (sz_c(c\, array), 0) \\ &= SZ_{\Uparrow}(\Delta \models c\, array \Uparrow S_T(c\, array)) \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash_c}(\Delta \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= SZ_{\Uparrow}(\Delta \models c\, array \Uparrow c'\, array) \end{aligned}$$

Lambda:

(1)

$$\begin{aligned} SZ_k(\Delta \models k) &= (sz_k(k), 0) \\ &< (sz_k(k) + sz_c(c), 0) \\ &= (sz_c(\lambda\alpha :: k.c), 0) \\ &= SZ_{\Uparrow}(\Delta \models \lambda\alpha :: k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

(2)

$$\begin{aligned} SZ_{\backslash_k}(\Delta \models k \backslash \kappa) &= (sz_k(k), 0) \\ &< (sz_c(\lambda\alpha :: k.c), 0) \\ &= SZ_{\Uparrow}(\Delta \models \lambda\alpha :: k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

(3)

$$\begin{aligned} SZ_{\Uparrow}(\Delta[\alpha :: \kappa] \models c \Uparrow \kappa') &= (sz_c(c), 0) \\ &< (sz_c(\lambda\alpha :: k.c), 0) \\ &= SZ_{\Uparrow}(\Delta \models \lambda\alpha :: k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

App:

(1)

$$\begin{aligned} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1\, c_2), 0) \\ &= SZ_{\Uparrow}(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{aligned}$$

(2)

$$\begin{array}{rcl} SZ_{\Downarrow}(\Delta \models c_2 \Downarrow \kappa_1) & = & (sz_c(c_2), 1) \\ & < & (sz_c(c_1\, c_2), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{array}$$

(3)

$$\begin{array}{rcl} SZ_{\backslash c}(\Delta \models c_2 \backslash c_2') & = & (sz_c(c_2), 0) \\ & < & (sz_c(c_1\, c_2), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{array}$$

Record: As before
Proj1: As before
Proj2:

(1)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2) & = & (sz_c(c), 0) \\ A & < & (sz_c(c) + 1, 0) \\ & = & (sz_c(c.1), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c.1 \Uparrow \{c'.1/\alpha\}\kappa_2) \end{array}$$

(2)

$$\begin{array}{rcl} SZ_{\backslash c}(\Delta \models c \backslash c') & = & (sz_c(c), 0) \\ & < & (sz_c(c.1), 0) \\ & = & SZ_{\Uparrow}(\Delta \models c.1 \Uparrow \{c'.1/\alpha\}\kappa_2) \end{array}$$

Let:

(1)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta \models c_1 \Uparrow \kappa_1) & = & (sz_c(c_1), 0) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (sz_c(\text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end}), 0) \\ & = & SZ_{\Uparrow}(\Delta \models \text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end} \Uparrow \{c_1'/\alpha\}\kappa_2) \end{array}$$

(2)

$$\begin{array}{rcl} SZ_{\Uparrow}(\Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2) & = & (sz_c(c_2), 0) \\ & < & (sz_c(c_1) + sz_c(c_2), 0) \\ & = & (sz_c(\text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end}), 0) \\ & = & SZ_{\Uparrow}(\Delta \models \text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end} \Uparrow \{c_1'/\alpha\}\kappa_2) \end{array}$$

(3)

$$\begin{array}{rcl} SZ_{\backslash c}(\Delta \models c_1 \backslash c_1') & = & (sz_c(c_1), 0) \\ & < & (sz_c(\text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end}), 0) \\ & = & SZ_{\Uparrow}(\Delta \models \text{let}\, \alpha = c_1 \,\text{in}\, c_2\, \text{end} \Uparrow \{c_1'/\alpha\}\kappa_2) \end{array}$$

□

Theorem 3. *The rules for kind-standardization, constructor standardization, kind synthesis, kind analysis, and kind well-formedness do not permit an infinite sequence of rule applications.*

Proof. By lemma **??** stated above, any infinite sequence of rule applications corresponds to an infinite descending chain of pairs of natural numbers ordered lexicographically, which contradicts the well-foundedness of $(N \times N, <)$.

□