Types for the (Eager) Functional Language

$$type ::= \mathbf{Bool} \mid \mathbf{Int} \mid type \to type \mid \prod(type, \ldots type) \mid \sum(type, \ldots type)$$

$$\frac{}{\pi \vdash \lfloor b \rfloor : \mathbf{Bool}} \quad b \in \mathbf{B}$$

$$\frac{}{\pi \vdash \lfloor n \rfloor : \mathbf{Int}} \quad n \in \mathbf{Z}$$

$$\frac{\pi \vdash e : \mathbf{Bool}}{\pi \vdash \neg e : \mathbf{Bool}}$$

$$\frac{\pi \vdash e : \mathbf{Int}}{\pi \vdash \text{-}e : \mathbf{Int}}$$

$$\frac{\pi \vdash e : \mathbf{Bool} \quad \pi \vdash e' : \theta \quad \pi \vdash e'' : \theta}{\pi \vdash \mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' : \theta}$$

$$\frac{}{\pi \vdash v : \theta} \quad v : \theta \text{ is in } \pi$$

$$\frac{\pi, v : \theta' \vdash e : \theta}{\pi \vdash \lambda v.\, e : \theta' \to \theta}$$

$$\frac{\pi \vdash e : \theta' \to \theta \quad \pi \vdash e' : \theta'}{\pi \vdash e\, e' : \theta}$$

$$\frac{\forall i \in 0\ \mathbf{to}\ n-1. \quad \pi \vdash e_i : \theta_i}{\pi \vdash \langle e_0, \ldots e_{n-1} \rangle : \prod(\theta_0, \ldots \theta_{n-1})}$$

$$\frac{\pi \vdash e : \prod(\theta_0, \ldots \theta_{n-1})}{\pi \vdash e.k : \theta_k} \quad \text{if } k < n$$

$$\frac{\pi \vdash e : \sum(\theta_0, \ldots \theta_{n-1}) \qquad \forall i \in 0\ \mathbf{to}\ n-1. \quad \pi \vdash e_i : \theta_i \to \theta}{\pi \vdash \mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots e_{n-1}) : \theta}$$

$$\frac{\pi \vdash e : \theta_k}{\pi \vdash @\, k\, e : \sum(\theta_0, \ldots \theta_{n-1})} \quad \text{if } k < n$$

Derived Forms

The unit type: $\mathbf{1} \stackrel{\text{def}}{=} \prod()$.

The empty tuple plays the role of the unit constant: $\vdash \langle\rangle : \prod()$.

The Boolean type and its operations can be derived:

$$
\begin{array}{rcl}
\mathbf{Bool} & \cong & \sum(\mathbf{1}, \mathbf{1}) \\
\mathbf{true} & \cong & @\,0\,\langle\rangle \\
\mathbf{false} & \cong & @\,1\,\langle\rangle \\
\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' & \cong & \mathbf{sumcase}\ e\ \mathbf{of}\ (\lambda v'.\, e',\ \lambda v''.\, e'') \\
 & & \quad \text{where } v' \notin FV(e'),\ v'' \notin FV(e'') \\
\neg e & \cong & \mathbf{if}\ e\ \mathbf{then\ false\ else\ true} \\
e' \wedge e'' & \cong & \mathbf{let}\ v \equiv e'\ \mathbf{in\ if}\ e''\ \mathbf{then}\ v\ \mathbf{else\ false}
\end{array}
$$

The empty type $\mathbf{Void} \stackrel{\text{def}}{=} \sum()$: no terms have this type in the eager language.

Derived Rules for the Derived Constructs

Extend contexts to allow bindings for patterns:

$$context ::= \ \mid context,\ pat : type$$

$$\frac{\pi,\ p : \theta' \vdash e : \theta}{\pi \vdash \lambda p.\, e : \theta' \to \theta}$$

$$\frac{\forall i \in 0 \textbf{ to } n-1.\quad \pi \vdash e_i : \theta_i \qquad \pi,\ p_0 : \theta_0, \ldots p_{n-1} : \theta_{n-1} \vdash e : \theta}{\pi \vdash \textbf{let } p_0 \equiv e_0, \ldots p_{n-1} \equiv e_{n-1} \textbf{ in } e : \theta}$$

$$\frac{\pi,\ p_0 : \theta_0, \ldots p_{n-1} : \theta_{n-1},\ \pi' \vdash e : \theta}{\pi,\ \langle p_0, \ldots p_{n-1} \rangle : \prod(\theta_0, \ldots \theta_{n-1}),\ \pi' \vdash e : \theta}$$

Example:

$$\frac{\dfrac{\dfrac{}{\vdash 1 : \textbf{Int}} \qquad \dfrac{}{\vdash 2 : \textbf{Int}}}{\vdash \langle 1, 2 \rangle : \prod(\textbf{Int}, \textbf{Int})} \qquad \dfrac{\dfrac{\ldots}{\mathsf{x} : \textbf{Int},\ \mathsf{y} : \textbf{Int} \vdash \mathsf{x+y} : \textbf{Int}}}{\langle \mathsf{x},\ \mathsf{y} \rangle : \prod(\textbf{Int}, \textbf{Int}) \vdash \mathsf{x+y} : \textbf{Int}}}{\vdash \textbf{let } \langle \mathsf{x},\ \mathsf{y} \rangle \equiv \langle 1, 2 \rangle \textbf{ in } \mathsf{x+y} : \textbf{Int}}$$

Soundness of the Type System

Lemma [Subject Reduction, big step]:
If $\vdash e : \theta$ and $e \Rightarrow z$, then $\vdash z : \theta$.

Lemma [Canonical Forms]:
If $\vdash z : \theta$, then

- if $\theta = \mathbf{Bool}$, then $z \in \{\mathbf{true}, \mathbf{false}\}$;
- if $\theta = \mathbf{Int}$, then $z = \lfloor n \rfloor$ for some $n \in \mathbf{Z}$;
- if $\theta = \theta' \to \theta''$, then $z = \lambda v.\, e$ for some v and e such that $v : \theta' \vdash e : \theta''$;
- if $\theta = \prod(\theta_0, \ldots \theta_{n-1})$, then $z = \langle z_0, \ldots z_{n-1} \rangle$ for some z_i such that $\vdash z_i : \theta_i$ for all $i \in 0$ **to** $n-1$;
- if $\theta = \sum(\theta_0, \ldots \theta_{n-1})$, then $z = @\ k\ z'$ where $k < n$ and $\vdash z' : \theta_k$.

Theorem [Soundness and Termination]:
If $\vdash e : \theta$, then there exists a canonical form z such that $e \Rightarrow z$ and $\vdash z : \theta$.

Recursion

In the untyped language we can define recursive functions using the (eager) fixed-point combinator Y_v.

But in a simple typing discipline Y_v is not typable.

One solution: a special construct for recursive definitions:

$$exp \;::=\; \ldots \mid \mathbf{letrec}\; var \equiv \lambda pat.\, exp, \ldots var \equiv \lambda pat.\, exp\; \mathbf{in}\; exp$$

with the reduction rule

$$\frac{(\lambda v_0.\, \ldots \lambda v_{n-1}.\, e)\,(\lambda u_0.\, e_0^*)\;\ldots\;(\lambda u_{n-1}.\, e_{n-1}^*) \;\Rightarrow\; z}{\mathbf{letrec}\; v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\; \mathbf{in}\; e \;\Rightarrow\; z}$$

where $e_i^* \stackrel{\text{def}}{=} \mathbf{letrec}\; v_0 \equiv \lambda u_0.\, e_0, \ldots v_{n-1} \equiv \lambda u_{n-1}.\, e_{n-1}\; \mathbf{in}\; e_i$

and $\{v_0, \ldots\} \cap \{u_0, \ldots\} = \{\}$

Syntactic sugar:

$$\mathbf{letrec}\; v_0\, p_0 \equiv e_0, \ldots\; \mathbf{in}\; e \stackrel{\text{def}}{=} \mathbf{letrec}\; v_0 \equiv \lambda p_0.\, e_0, \ldots\; \mathbf{in}\; e$$

Typing Lists

A list of integers can have all of the shapes

$$\begin{array}{ll}
\vdash @\,0\ \langle\rangle & : \mathbf{1} + \theta_0 \\
\vdash @\,1\ \langle \lfloor n_1 \rfloor, @\,0\ \langle\rangle\rangle & : \theta_1 + \mathbf{Int} \times (\mathbf{1} + \theta_0) \\
\vdash @\,1\ \langle \lfloor n_2 \rfloor, @\,1\ \langle \lfloor n_1 \rfloor, @\,0\ \langle\rangle\rangle\rangle & : \theta_2 + \mathbf{Int} \times (\theta_1 + \mathbf{Int} \times (\mathbf{1} + \theta_0)) \\
\ldots &
\end{array}$$

where $\theta + \theta' \stackrel{\text{def}}{=} \sum(\theta, \theta')$ and $\theta \times \theta' \stackrel{\text{def}}{=} \prod(\theta, \theta')$.

A type **intlist** of all lists of integers must match all of these types,
so we have to unify these types, hence

$$\begin{array}{rcl}
\theta_1 & = & \mathbf{1} \\
\theta_2 & = & \mathbf{1} \\
\ldots & &
\end{array}$$

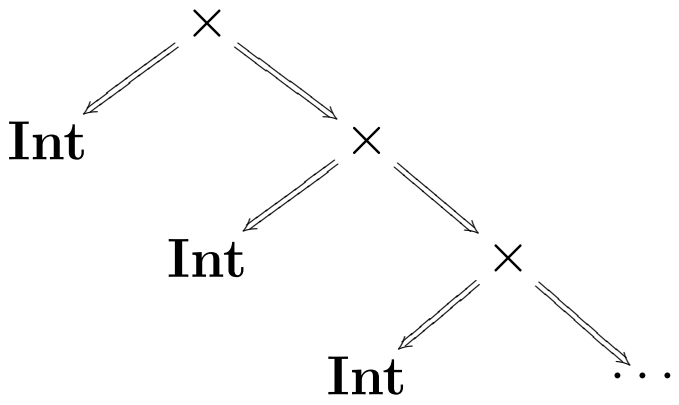
but what about θ_0? It must be the same as $\mathbf{Int} \times (\mathbf{1} + \theta_0)$
— impossible for finite types.

Recursive Types

To find a type for lists, we have to allow infinite types.

In many cases it is sufficient to consider only regular types.

A regular tree is a tree with finitely many (distinct) subtrees:



Syntax: $\mu\alpha.\,\mathbf{Int} \times \alpha$ used to denote this tree; α is a type variable.

$$type ::= \ldots \mid tyvar \mid \mu tyvar.\, type$$

A regular recursive type $\mu\tau.\,\theta$ is syntactically equivalent to its unfolding:

$$\mu\tau.\,\theta \;=\; \theta/\tau \to \mu\tau.\,\theta$$

$$\mu\alpha.\,\mathbf{Int} \times \alpha \;=\; \mathbf{Int} \times \mu\alpha.\,\mathbf{Int} \times \alpha \;=\; \mathbf{Int} \times \mathbf{Int} \times \mu\alpha.\,\mathbf{Int} \times \alpha \;=\; \ldots$$

The Recursive Type of Lists

$$\begin{array}{ll}
\vdash @\,0\,\langle\rangle & : \mathbf{1} + \theta_0 \\
\vdash @\,1\,\langle \lfloor n_1 \rfloor, @\,0\,\langle\rangle\rangle & : \mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \theta_0) \\
\vdash @\,1\,\langle \lfloor n_2 \rfloor, @\,1\,\langle \lfloor n_1 \rfloor, @\,0\,\langle\rangle\rangle\rangle & : \mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \theta_0)) \\
\ldots &
\end{array}$$

The requirement $\theta_0 = \mathbf{Int} \times (\mathbf{1} + \theta_0)$ is satisfied if $\theta_0 = \mu\alpha.\, \mathbf{Int} \times (\mathbf{1} + \alpha)$:

$$\theta_0 \;=\; \mu\alpha.\, \mathbf{Int} \times (\mathbf{1} + \alpha) \;=\; (\mathbf{Int} \times (\mathbf{1} + \alpha))/\alpha \to \theta_0 \;=\; \mathbf{Int} \times (\mathbf{1} + \theta_0)$$

So lists of integers can be given the type

$$\begin{array}{rrl}
 & \mathbf{intlist} & = \mathbf{1} + \mu\alpha.\, \mathbf{Int} \times (\mathbf{1} + \alpha) \\
 & & = \mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \ldots)) \\
 & & = \mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \mathbf{Int} \times (\mathbf{1} + \ldots)) \\
 & & = \mu\beta.\, \mathbf{1} + \mathbf{Int} \times \beta \\
\text{satisfying} & \mathbf{intlist} & = \mathbf{1} + \mathbf{Int} \times \mathbf{intlist}
\end{array}$$

i.e. "an **intlist** is either a 0-tagged empty tuple,
or a 1-tagged pair of an in integer and an **intlist**."

Recursive Types Break Strong Normalization

With recursive types we can give a type to the self-applicator Δ:

$$\dfrac{\dfrac{}{\mathsf{x} : \mu\alpha.\,\alpha \to \theta \vdash \mathsf{x} : (\mu\alpha.\,\alpha \to \theta) \to \theta} \quad \dfrac{}{\mathsf{x} : \mu\alpha.\,\alpha \to \theta \vdash \mathsf{x} : \mu\alpha.\,\alpha \to \theta}}{\dfrac{\mathsf{x} : \mu\alpha.\,\alpha \to \theta \vdash \mathsf{x}\mathsf{x} : \theta}{\vdash \lambda\mathsf{x}.\,\mathsf{x}\mathsf{x} : (\mu\alpha.\,\alpha \to \theta) \to \theta}}$$

for any type θ, since $\mu\alpha.\,\alpha \to \theta = (\mu\alpha.\,\alpha \to \theta) \to \theta$.

Then the non-terminating term $\Omega = \Delta\,\Delta$ can be given any type θ:

$$\dfrac{\dfrac{\ldots}{\vdash \Delta : (\mu\alpha.\,\alpha \to \theta) \to \theta} \quad \dfrac{\ldots}{\vdash \Delta : \mu\alpha.\,\alpha \to \theta}}{\vdash \Delta\,\Delta : \theta}$$

Typing the Normal-Order Fixed-Point Combinator

$$\dfrac{\dfrac{\dots}{\mathsf{f} : \theta \to \theta \vdash \Delta : \theta_0 \to \theta} \quad \dfrac{\dfrac{\dfrac{}{\pi \vdash \mathsf{f} : \theta \to \theta} \quad \dfrac{\dfrac{}{\pi \vdash \mathsf{x} : \theta_0 \to \theta} \quad \dfrac{}{\pi \vdash \mathsf{x} : \theta_0}}{\pi \vdash \mathsf{x}\,\mathsf{x} : \theta}}{\pi \vdash \mathsf{f}\,(\mathsf{x}\,\mathsf{x}) : \theta}}{\mathsf{f} : \theta \to \theta \vdash \lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x}) : \theta_0 \to \theta}}{\mathsf{f} : \theta \to \theta \vdash \Delta\,(\lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x})) : \theta}$$

$$\vdash \lambda \mathsf{f}.\, \Delta\,(\lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x})) : (\theta \to \theta) \to \theta$$

where $\theta_0 = \mu\alpha.\, \alpha \to \theta$

$\pi = \mathsf{f} : \theta \to \theta, \mathsf{x} : \theta_0$

Thus $\vdash Y : (\theta \to \theta) \to \theta$ (recall that $\mathbf{Y}_D \in [[D \to D] \to D]$).

Typing the Eager Fixed-Point Combinator

$$\dfrac{\dfrac{\dots}{\mathsf{f} : \theta \to \theta \vdash \Delta : \theta_0 \to \theta} \quad \dfrac{\dfrac{\dfrac{}{\pi \vdash \mathsf{f} : \theta \to \theta} \quad \dfrac{\dfrac{\dfrac{}{\pi' \vdash \mathsf{x} : \theta_0 \to \theta} \quad \dfrac{}{\pi' \vdash \mathsf{x} : \theta_0}}{\pi' \vdash \mathsf{x}\,\mathsf{x} : \theta} \quad \dfrac{}{\pi' \vdash \mathsf{y} : \theta'}}{\pi' \vdash \mathsf{x}\,\mathsf{x}\,\mathsf{y} : \theta''}}{\pi \vdash \lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{x}\,\mathsf{y} : \theta}}{\pi \vdash \mathsf{f}\,(\lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{x}\,\mathsf{y}) : \theta}}{\mathsf{f} : \theta \to \theta \vdash \lambda \mathsf{x}.\, \mathsf{f}\,(\lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{x}\,\mathsf{y}) : \theta_0 \to \theta}}{\mathsf{f} : \theta \to \theta \vdash \Delta\,(\lambda \mathsf{x}.\, \mathsf{f}\,(\lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{x}\,\mathsf{y})) : \theta}}{\vdash \lambda \mathsf{f}.\, \Delta\,(\lambda \mathsf{x}.\, \mathsf{f}\,(\lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{x}\,\mathsf{y})) : (\theta \to \theta) \to \theta}$$

where $\theta = \theta' \to \theta''$

$\theta_0 = \mu\alpha.\, \alpha \to \theta$

$\pi = \mathsf{f} : \theta \to \theta, \mathsf{x} : \theta_0$

$\pi' = \pi, \mathsf{y} : \theta'$

Thus $\vdash Y_v : ((\theta' \to \theta'') \to \theta' \to \theta'') \to \theta' \to \theta''$.