1 Function and Unit Types

We’re going to start adding types to the (formerly untyped) lambda calculus. The first type we’ll add is a type of functions $\tau_1 \rightarrow \tau_2$, which is a function from arguments of type $\tau_1$ to results of type $\tau_2$. The arrow will associate to the right, so the type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ means a function that takes a $\tau_1$ and returns a function that takes a $\tau_2$ and returns a $\tau_3$, which is how we implemented multi-argument functions in lambda calculus.

The types $\tau_1$ and $\tau_2$ can themselves be functions, but this recursion needs to bottom out somewhere, so we need another type. We’ll add a “base type” unit (sometimes notated 1) with one value () (in C, you might think of this as the type “void”). Expressions $e$ are the same as lambda terms but now we actually say what the type of a function argument is (like we do in C, Java, etc.).

We’ll keep adding to this, but this is the language for now:

$$
\begin{align*}
\text{Expressions} \quad e & ::= x \mid () \mid \lambda x : \tau.e \mid e \; e \\
\text{Types} \quad \tau & ::= \text{unit} \mid \tau \rightarrow \tau
\end{align*}
$$

And here are the static and dynamic rules for now:

$$
\begin{align*}
\Gamma(x) = \tau & \quad (\text{VAR}) \\
\Gamma \vdash x : \tau & \quad (\text{I}) \\
\Gamma \vdash () : \text{unit} & \quad (\text{unit-I}) \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2 : \tau_1 & \quad (\rightarrow-E) \\
\Gamma, x : \tau \vdash e : \tau' & \quad (\rightarrow-I) \\
\Gamma \vdash \lambda x : \tau.e : \tau \rightarrow \tau & \quad (\lambda-I) \\
\Gamma \vdash \lambda x : \tau.e : \tau \rightarrow \tau & \quad (\lambda-I) \\
\Gamma \vdash e_1 \rightarrow e'_1 & \quad (\rightarrow-\text{val}) \\
\Gamma \vdash e_2 \rightarrow e'_2 & \quad (S-2) \\
\Gamma \vdash (\lambda x : \tau.e) e_2 \rightarrow (\lambda x : \tau.e) e'_2 & \quad (S-2) \\
\Gamma \vdash e_1 \rightarrow e'_1 & \quad (S-1) \\
\Gamma \vdash e_2 \rightarrow e'_2 & \quad (S-1) \\
\Gamma \vdash v \rightarrow [v/x]e & \quad (S-3)
\end{align*}
$$

Note that for the statics, instead of using numbers, we’re now labeling the rules with the type they deal with (→ or unit) and whether they Introduce or Eliminate expressions of that type.

Example.

$$(\lambda x : \text{unit}.x) ()$$

$$
\begin{align*}
&x : \text{unit} \vdash x : \text{unit} \quad (\text{VAR}) \\
\Gamma \vdash \lambda x : \text{unit}.x : \text{unit} \rightarrow \text{unit} & \quad (\rightarrow-I) \\
\Gamma, () \vdash () : \text{unit} & \quad (\text{unit-I}) \\
\Gamma \vdash (\lambda x : \text{unit}.x) () : () & \quad (\rightarrow-E)
\end{align*}
$$

$$
\begin{align*}
&\Gamma \vdash (\lambda x : \text{unit}.x) : \text{unit} \\
&\therefore \Gamma \vdash () : \text{unit}
\end{align*}
$$
Example.

\((\lambda f : \text{unit} \to \text{unit} . f ()) \ (\lambda x : \text{unit}.x) : \text{unit}\)

\((\lambda f : \text{unit} \to \text{unit} . f ()) \ (\lambda x : \tau .x)\) for any \(\tau \neq \text{unit}\) isn’t well-typed: the type annotations matter!

## 2 Products

**Expressions**  
\( e ::= \cdots | (e,e) \mid \text{fst } e \mid \text{snd } e \)

**Types**  
\( \tau ::= \cdots | \tau \times \tau \)

**Examples**  
(Let’s assume we also have \(\text{int}\) and \(\text{string}\) as base types for these.)

\[
\begin{align*}
(\text{T}, \text{T}) & \quad : \text{int } \times \text{int} \\
(\text{T}, \text{“Hi”}) & \quad : \text{int } \times \text{string} \\
(\text{“Hi”}, \text{T}) & \quad : \text{string } \times \text{int} \\
((), \text{T}) & \quad : \text{unit } \times \text{int} \\
\lambda x : \text{unit}.\lambda y : \text{int}.(x,y) & \quad : \text{unit } \to \text{int } \to (\text{unit } \times \text{int}) \\
((\lambda x : \text{unit}.x), \text{T}) & \quad : ((\text{unit } \to \text{unit}), \text{int}) \\
\lambda x : \text{int } \times \text{string}.\text{fst } x & \quad : \text{int}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma & \vdash e_2 : \tau_2 \quad (\times I) \quad \Gamma & \vdash e : \tau_1 \times \tau_2 \quad (\times E_1) \\
\Gamma & \vdash \text{fst } e : \tau_1 \quad (\times E_2)
\end{align*}
\]

\[
\begin{align*}
\text{val } v_1 & \quad \text{val } v_2 \\
(\text{Val}, \text{Val}) & \quad \text{val } (v_1, v_2) \\
\text{Val} & \quad \text{val } \text{e} \\
\text{Val} & \quad \text{val } \text{e}'
\end{align*}
\]

\[
\begin{align*}
\text{Val} & \quad \text{val } v_1 \quad \text{Val} \quad \text{val } v_2 \\
\text{Val} & \quad \text{val } \text{e} \\
\text{Val} & \quad \text{val } \text{e}'
\end{align*}
\]

\[
\begin{align*}
\text{Val} & \quad \text{val } \text{e} \quad \text{Val} \quad \text{val } \text{e}'
\end{align*}
\]

## 3 Sums

**Expressions**  
\( e ::= \cdots | \text{inl } e | \text{inr } e | \text{case } e \text{ of } \{x.e; y.e\} \)

**Types**  
\( \tau ::= \tau + \tau \)

\(\text{int } + \text{string} \): can be an int or a string (but you have to say which one)

**Examples.**

\[
\begin{align*}
\text{inl } \text{T} & \quad : \text{int } + \text{string} \\
\text{inr } \text{“Hi”} & \quad : \text{int } + \text{string} \\
\text{case } \text{inl } \text{T} \text{ of } \{x.x; y.y\} & \quad : \text{int} \\
\text{case } \text{inr } \text{“Hi”} \text{ of } \{x.x; y.y\} & \quad : \text{int} \\
\lambda x : \text{int } + \text{string}.\text{case } x \text{ of } \{x.x; y.y\} & \quad : (\text{int } + \text{string}) \to \text{int}
\end{align*}
\]
\[\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \text{inl} \ e : \tau_1 + \tau_2 \quad \Gamma \vdash \text{inr} \ e : \tau_1 + \tau_2\]

\[\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of } \{x.e_1; y.e_2\} : \tau} \quad (+E)\]

\[\frac{v \text{ val}}{\text{val} v \vdash \text{inl} v} \quad \text{(V-4)}\]
\[\frac{v \text{ val}}{\text{val} v \vdash \text{inr} v} \quad \text{(V-5)}\]

\[\text{e \mapsto e'} \quad \text{(S-10)}\]
\[\text{inl} \ e \mapsto \text{inl} e' \quad \text{(S-11)}\]
\[\frac{v \text{ val}}{\text{case } \text{inl} \ v \text{ of } \{x.e_1; y.e_2\} \mapsto [v/x]e_1} \quad \text{(S-13)}\]
\[\frac{v \text{ val}}{\text{case } \text{inr} \ v \text{ of } \{x.e_1; y.e_2\} \mapsto [v/y]e_2} \quad \text{(S-14)}\]

Example: Options (like null pointer)
\[
\text{int option}\triangleq\text{int + (}
\]
\[
\text{case e of } \{x.x; y.y\}
\]

4 Type Safety

Lemma 1 (Canonical Forms). 1. If e val and \(\bullet \vdash e : \tau_1 \times \tau_2\), then \(e = (v_1, v_2)\) where \(v_1\) val and \(v_2\) val.

2. If e val and \(\bullet \vdash e : \tau_1 + \tau_2\), then \(e = \text{inl} \ v\) where \(v\) val or \(e = \text{inr} \ v\) where \(v\) val.

Theorem 1 (Progress). If \(\bullet \vdash e : \tau\) then either e val or there exists e' such that e \(\mapsto e'\).

Proof. By induction on the derivation of \(\bullet \vdash e : \tau\).

- (+E). Then e = case e_1 of \{x.e_2; y.e_3\} and \(\bullet \vdash e_1 : \tau_1 + \tau_2\) and x : \(\tau_1\vdash e_2 : \tau\) and y : \(\tau_2\vdash e_3 : \tau\). By induction, e_1 val or there exists e_1 \(\mapsto e_1'\). Then, by canonical forms, e_1 = inl v or e_2 = inr v, and e steps by (S-13) or (S-14).

Theorem 2 (Preservation). If \(\bullet \vdash e : \tau\) and e \(\mapsto e'\) then \(\bullet \vdash e' : \tau\).

Proof. By induction on the derivation of e \(\mapsto e'\).

- (S-1). Then e = c_1 e_2. By inversion, \(\bullet \vdash e_1 : \tau' \rightarrow \tau\). By induction, \(\bullet \vdash e_1' : \tau' \rightarrow \tau\). Apply \(\rightarrow\)-E.

- (S-2). Similar.

- (S-3). Then e = (\lambda x : \tau'. e_0) v and e' = [v/x]e. By inversion on \(\rightarrow\)-E, \(\bullet \vdash \lambda x : \tau'. e_0 : \tau' \rightarrow \tau\) and \(\bullet \vdash v : \tau'\). By inversion on \(\rightarrow\)-I, x : \(\tau'\vdash e_0 : \tau\). By substitution, \(\bullet \vdash [v/x]e_0 : \tau\).

- (S-7). Then e = \text{fst} (e', v_2). By inversion on \(\times\)-E1, \(\bullet \vdash \langle e', v_2 \rangle : \tau \times \tau_2\). By inversion on \(\times\)-I, \(\bullet \vdash e' : \tau\).

- (S-13). Then e = \text{case inl} v of \{x.e_1; y.e_2\} and v val and e' = [v/x]e_1. By inversion, and \(\bullet \vdash \text{inl} v : \tau_1 + \tau_2\) and x : \(\tau_1\vdash e_1 : \tau\). By another inversion, \(\bullet \vdash v : \tau_1\). By substitution, \(\bullet \vdash [v/x]e_1 : \tau\).
Fun fact If $\vdash e : \tau$ then there exists $v$ such that $v \text{val}$ and $e \mapsto^* v$. (We won’t prove it though, that proof is actually pretty tricky and uses some techniques we probably won’t see in the class.)

Remember that this wasn’t true for the untyped lambda calculus. So why is it true now that we’ve added types? Remember our self-application trick. Let’s try to figure out the type of $(\lambda x.x \ x) \ (\lambda x.x \ x)$. There’s no type annotation, so let’s just say the type of $x$ is $\tau$ and we’ll figure it out later. We’ll also say the return type of each function is $\tau'$.

\[
\begin{array}{c}
\frac{x : \tau \vdash x : \tau}{x : \tau \vdash x : \tau} \quad \frac{x : \tau \vdash x : \tau'}{\bullet \vdash \lambda x.x : \tau \rightarrow \tau'}
\end{array}
\]

5 Observations

Booleans

\begin{align*}
\text{true} & \triangleq \text{inl} () \\
\text{false} & \triangleq \text{inr} () \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} & \triangleq \text{case } e_1 \text{ of } \{ x.e_2 ; y.e_3 \}
\end{align*}

- The type of Booleans is sometimes called 2 because it has 2 elements.
- We write $\cong$ to mean that two types are “equal” (really isomorphic, but we’re not going to go into the definition of that).
- So 2 $\cong 1 + 1$!
- This isn’t a coincidence. Think about how many elements, e.g., the type $2 \times 2$ has (i.e., how many different pairs of Booleans there are).

More general sums and products

- **Binary products** $\tau_1 \times \tau_2$:
  - 1 intro form (way to create)—pair
  - 2 elim forms (things to do with them)—projections $\text{fst}, \text{snd}$
- **n-ary products** $\tau_1 \times \cdots \times \tau_n$:
  - 1 intro form (way to create)—“tuple”
  - $n$ elim forms (things to do with them)—projections $\pi_i$
  - (We can also just encode this as nested binary products: $\tau_1 \times (\tau_2 \times (\cdots \times (\tau_{n-1} \times \tau_n))))$)
- **Binary sums** $\tau_1 + \tau_2$:
  - 2 intro forms (inl, inr)
  - 1 elim form (case)
- **n-ary sums** $\tau_1 + \cdots + \tau_n$:
  - $n$ intro forms (“injections”)
  - 1 elim form (case)
- 0-ary (“nullary”) product?
Inference Rules (including void)

- 1 intro form (way to create)—“tuple”
- 0 elim forms—nothing to do with it
- Unit!

- Nullary sum?
  - 0 intro forms
  - 1 thing to do
  - void

6 Inference Rules (including void)

\[
\begin{align*}
\Gamma(x) &= \tau \quad \text{(VAR)} \\
\Gamma \vdash x : \tau \\
\Gamma \vdash () : \text{unit} \quad \text{(unit-I)} \\
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 \rightarrow e_2 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e \rightarrow e' : \tau \\
\Gamma \vdash \text{case } e \text{ of } \{ x.e_1; y.e_2 \} : \tau \\
\Gamma \vdash \text{void} : \tau \\
\Gamma \vdash \text{abort } e : \tau \\
\end{align*}
\]