Scribe: Andrew Myers (based on notes by Yan Zhang, Antonio Montalban) Lecturer: Andrew Myers

1 Progress Lemma

To finish the proof of Soundness for the typed lambda calculus, we need to prove Progress. The Progress Lemma captures the idea that we cannot get stuck when evaluating a well-formed expression.

Progress Lemma: $\vdash e : \tau \Rightarrow e \in \mathit{Value} \vee \exists e'.e \longrightarrow e'$
Proof: We use induction on the typing derivation of e. Remember the definition of an expression in $\lambda^{\rightarrow}$:

$$e ::= b \mid x \mid \lambda x{:}\tau.\, e \mid e_0\ e_1$$

So we have four cases:

- Case $e = b$: We have that $b \in \mathit{Value}$.
- Case $e = x$: This case is not possible because we would have $\vdash x : \tau$ and from the empty environment we cannot assign any type to x.
- Case $e = \lambda x \in \tau_0\,.\,e_1$: We have that $e \in \mathit{Value}$.
- Case $e = e_0\,e_1$: We know that there is a typing derivation for $\vdash e_0\,e_1 : \tau$ and this derivation must have the form:

$$\frac{\vdash e_0 : \tau' \quad \vdash e_1 : \tau' \rightarrow \tau}{\vdash e_0\,e_1 : \tau}$$

  By the induction hypothesis, $e_0 \in \mathit{Value} \vee \exists e_0'.e_0 \longrightarrow e_0'$ and $e_1 \in \mathit{Value} \vee \exists e_1'.e_1 \longrightarrow e_1'$. We have four possibilities now:

  - Both e_0 and e_1 are values. Since e_0 is a value with an arrow type, it has to be an abstraction. Say e_0 is $\lambda x \in \tau'\,.\,e_2$ and e_1 is some value v. Then

$$e = (\lambda x \in \tau'\,.\,e_2)v \longrightarrow e_2\{v/x\}$$

    so, $e' = e_2\{v/x\}$ as desired.
  - e_0 is not a value. Then $\exists e_0'.e_0 \longrightarrow e_0'$ and we have

$$\frac{e_0 \longrightarrow e_0'}{e_0\,e_1 \longrightarrow e_0' e_1}$$

  - e_0 is some value v, but e_1 is not a value. Then $\exists e_1'.e_1 \longrightarrow e_1'$ and we have

$$\frac{e_1 \longrightarrow e_1'}{v\,e_1 \longrightarrow v\,e_1'}$$

And this finishes the proof.

2 $\lambda^{\rightarrow *+}$

In comparison to UML, the language $\lambda^{\rightarrow}$ has a lot of stuff missing. Let's add some of this stuff to the language in a slightly simplified form. We extend $\lambda^{\rightarrow}$ to $\lambda^{\rightarrow *+}$ as follows:

$$e ::= \ldots \mid (e_0, e_1) \mid \mathsf{left}\ e \mid \mathsf{right}\ e \mid \mathsf{case}\ e_0\ \mathsf{of}\ e_1|e_2 \mid \mathsf{inl}_{\tau_1+\tau_2} e \mid \mathsf{inr}_{\tau_1+\tau_2} e$$

We also extend our values

$$v ::= \lambda x \in \tau \,.\, e \;\mid\; (v_0, v_1) \;\mid\; \mathsf{inl}_{\tau_1+\tau_2} v \;\mid\; \mathsf{inr}_{\tau_1+\tau_2} v$$

The set of types is defined by

$$\tau ::= B \;\mid\; \tau_0 \to \tau_1 \;\mid\; \tau_0 * \tau_1 \;\mid\; \tau_0 + \tau_1$$

where $\tau_0 * \tau_1$ and $\tau_0 + \tau_1$ are the product type and sum type of τ_0 and τ_1.

Now we define the operational semantics. We start by extending our evaluation contexts:

$$E[\cdot] ::= \ldots \;\mid\; ([\cdot], e) \;\mid\; (v, [\cdot]) \;\mid\; \mathsf{left}\ [\cdot] \;\mid\; \mathsf{right}\ [\cdot] \;\mid\; \mathsf{case}\ [\cdot]\ \mathsf{of}\ e_1|e_2 \;\mid\; \mathsf{inl}_{\tau_1+\tau_2}[\cdot] \;\mid\; \mathsf{inr}_{\tau_1+\tau_2}[\cdot]$$

and then we define the rules. We have the usual rule

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

where the reductions are

- $(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}$
- $\mathsf{left}\ (v_0, v_1) \longrightarrow v_0$
- $\mathsf{right}\ (v_0, v_1) \longrightarrow v_1$
- $\mathsf{case}\ (\mathsf{inl}_{\tau_1+\tau_2} v)\ \mathsf{of}\ e_1|e_2 \longrightarrow e_1\, v$
- $\mathsf{case}\ (\mathsf{inr}_{\tau_1+\tau_2} v)\ \mathsf{of}\ e_1|e_2 \longrightarrow e_2\, v$

Observe that we have introduction forms (constructors) and elimination forms (destructors). The constructors construct elements of more complex types from simpler ones. For example $(\cdot, \cdot)$ constructs elements of type $\tau_0 * \tau_1$ from two elements, one of type τ_0 and the other of type τ_1. The other constructors are the abstraction and the inclusions inl and inr. The destructors are the application, case, left and right operations. A redex is an expression where a constructor and its corresponding destructor collide.

Observe that we do not need booleans in $\lambda^{\to *+}$. They can encoded as follows:

- $[\![\mathsf{bool}]\!] = 1 + 1$
- $[\![\mathsf{true}]\!] = \mathsf{inl}_{1+1}\mathsf{unit}$
- $[\![\mathsf{false}]\!] = \mathsf{inr}_{1+1}\mathsf{unit}$
- $[\![\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2]\!] = \mathsf{case}\ e_0\ \mathsf{of}\ \lambda x{:}1.\, [\![e_1]\!] | \lambda x{:}1.\, [\![e_2]\!]$ where x is a fresh variable.

3 Typing Rules

Now we give the typing rules for typed lambda calculus:

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0, e_1) : \tau_0 * \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathsf{left}\ e : \tau_0} \qquad \frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathsf{right}\ e : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2} e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2} e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_2 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_1 : \tau_1 \to \tau_3 \quad \Gamma \vdash e_0 : \tau_1 + \tau_2}{\Gamma \vdash \mathsf{case}\ e_0\ \mathsf{of}\ e_1|e_2 : \tau_3}$$

4 Denotational semantics

The denotational semantics for type domains are as follows:

$$\begin{aligned}\mathcal{T}[\![\tau_1 \to \tau_2]\!] &= \mathcal{T}[\![\tau_2]\!]^{\mathcal{T}[\![\tau_1]\!]} \\ \mathcal{T}[\![\tau_1 * \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!] \\ \mathcal{T}[\![\tau_1 + \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!]\end{aligned}$$

In the right-hand side, $\times$ and $+$ mean mathematical product and disjoint union. Now we give the semantic function for this language:

$$\rho \models \Gamma \Rightarrow \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho \ \in \mathcal{T}[\![\tau]\!]$$

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash (e_0, e_1) : \tau_0 * \tau_1]\!] =& \langle \mathcal{C}[\![\Gamma \vdash e_0 : \tau_0]\!]\rho, \mathcal{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho\rangle \in \mathcal{T}[\![\tau_0 * \tau_1]\!]\rho \\ \mathcal{C}[\![\Gamma \vdash \mathsf{left}\ e : \tau_0]\!]\rho =& \pi_1(\mathcal{C}[\![\Gamma \vdash e : \tau_0 * \tau_1]\!]\rho) \in \mathcal{T}[\![\tau_0]\!] \\ \mathcal{C}[\![\Gamma \vdash \mathsf{right}\ e : \tau_1]\!]\rho =& \pi_2(\mathcal{C}[\![\Gamma\ \vdash e : \tau_0 * \tau_1]\!]\rho) \in \mathcal{T}[\![\tau_1]\!] \\ \mathcal{C}[\![\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2} e : \tau_1 + \tau_2]\!]\rho =& in_1(\mathcal{C}[\![\Gamma \vdash e : \tau_1]\!]\rho) \in \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!] \\ \mathcal{C}[\![\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2} e : \tau_1 + \tau_2]\!]\rho =& in_2(\mathcal{C}[\![\Gamma \vdash e : \tau_2]\!]\rho) \in \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!] \\ \mathcal{C}[\![\Gamma \vdash \mathsf{case}\ e_0\ \mathsf{of} e_1 | e_2]\!]\rho =& case\ \mathcal{C}[\![\Gamma \vdash e_0 : \tau_1 + \tau_2]\!]\rho\ of \\ & \quad in_1(x_1).(\mathcal{C}[\![\Gamma \vdash e_1 : \tau_1 \to \tau_3]\!]\rho)x_1 \\ & \mid \quad in_2(x_2).(\mathcal{C}[\![\Gamma \vdash e_2 : \tau_2 \to \tau_3]\!]\rho)x_2 \\ & end \in \mathcal{T}[\![\tau_3]\!]\end{aligned}$$

Notice that the sums and products we gave above can be extended to arbitrary tuple and variant types. This can be also obtained by the following desugaring:

$$\begin{aligned}\tau_1 * ... * \tau_n &= \tau_1 * (\tau_2 * ... * \tau_n) \\ \langle e_1, ..., e_n\rangle &= \langle e_1, \langle e_2, ..., e_n\rangle\rangle\end{aligned}$$

Variants (such as datatypes) can be desugared into sums similarly.

5 Recursion

To make the language Turing-equivalent, extend the language as follows:

$$e ::= ... \ \mid \ \mathsf{rec}\ y : \tau \to \tau'.(\lambda x.e)$$

With this extension, the language becomes essentially the language PCF. (Note that Pierce introduces a somewhat more general language-level fix operator.)

$$\frac{\Gamma, x : \tau,\ y : \tau \to \tau' \quad \vdash e : \tau'}{\Gamma \vdash \mathsf{rec}\ y : \tau \to \tau'.(\lambda x.e) : \tau \to \tau'}$$

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash \mathsf{rec}\ y : \tau \to \tau'.(\lambda x\ e) : \tau \to \tau']\!]\rho \ =\ & fix\ \lambda f \in \mathcal{T}[\![\tau \to \tau']\!]. \\ & \lambda v \in \mathcal{T}[\![\tau]\!]\,.\,\mathcal{C}[\![\Gamma, x : \tau,\ y : \tau \to \tau'\ \vdash e : \tau']\!]\rho[x \mapsto v, y \mapsto f]\end{aligned}$$

Because of the fixed point, the domain $\mathcal{T}[\![\tau \to \tau']\!]$ must be a pointed cpo. So we add $\bot$ as a possible function result and only allow continuous functions too:

$$\mathcal{T}[\![\tau \to \tau']\!] = [\mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]_\bot]$$

Further, the meaning function now gives elements from a lifted domain $\mathcal{T}[\![\tau]\!]_\bot$ instead of from $\mathcal{T}[\![\tau]\!]$:

$$\rho \models \Gamma \Rightarrow \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho \in \mathcal{T}[\![\tau]\!]_\bot$$

By adding recursion to this language and making the domains pointed cpos, we can write non-terminating programs in this language, and in fact the language will be Turing complete if we have a few operations on numbers. We also have to make a few changes to the definition of $\mathcal{C}[\![\cdot]\!]$ using the let construct from the metalanguage to handle the $\bot$'s.

Example:

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash e_0\ e_1 : \tau']\!]\rho = \quad & \mathsf{let} f = \mathcal{C}[\![\Gamma \vdash e_0 : \tau \to \tau']\!]\rho. \\ & \mathsf{let}\ v = \mathcal{C}[\![\Gamma \vdash e_1 : \tau]\!]\rho. f(v)\end{aligned}$$