References

- So far the languages that we've looked at do not include operations that change state.
- For example, we did not include a way to change the value of the first element of a pair.
- Today we add "references" (pointers) to the simply typed lambda calculus.
- The new operations are:
  - allocate a location in memory and initialize it with a value, such as `new int(3)` in C++
  - read the value from a memory location, such as `*p` in C++
  - change the value in a memory location, such as `*p = 6` in C++

References: Syntax and Type Rules

$$
\begin{array}{lcll}
e & ::= & \ldots & \\
 & | & \mathit{ref}\ e & \text{allocate} \\
 & | & !e & \text{dereference} \\
 & | & e := e & \text{update} \\
v & ::= & \ldots \mid l & \text{locations} \\
T & ::= & \ldots \mid \mathit{Ref}\ T &
\end{array}
$$

Type rules:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathit{ref}\ e : \mathit{Ref}\ T} \qquad \frac{\Gamma \vdash e : \mathit{Ref}\ T}{\Gamma \vdash !e : T}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Ref}\ T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathit{Unit}}$$

References: Operational Semantics

Change to the framework: need to model the heap. Abstractly, we can use a function from locations to values, denoted by μ.

$$\boxed{e \mid \mu \longrightarrow e \mid \mu}$$

$$\frac{l \notin dom(\mu)}{ref\ v \mid \mu \longrightarrow l \mid [l \mapsto v]\mu} \qquad \frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu}$$

$$\frac{}{l := v \mid \mu \longrightarrow unit \mid [l \mapsto v]\mu}$$

Evaluation contexts:

$$E ::= \ldots \mid ref\ E \mid !E \mid E := e \mid v := E$$

Towards Type Safety

- If this language is to satisfy "progress", then all intermediate steps need to be well-typed.
- With the addition of references, we now have a new kind of value: locations, that can appear during intermediate steps.
- We introduce a *store typing* Σ to keep track of the type allowed at each location.

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ T}$$

Definition

A store μ is well typed, written $\Gamma \mid \Sigma \vdash \mu$, if $\mathit{dom}(\mu) = \mathit{dom}(\Sigma)$ and for every $l \in \mathit{dom}(\mu)$, $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$.

Towards Type Safety

Lemma (Subject Reduction)

If $\Gamma \mid \Sigma \vdash e : T$ *and* $\Gamma \mid \Sigma \vdash \mu$ *and* $e \mid \mu \longrightarrow e' \mid \mu'$ *then for some* Σ' *where* $\Sigma \subseteq \Sigma'$, $\Gamma \mid \Sigma' \vdash e' : T$ *and* $\Gamma \mid \Sigma' \vdash \mu'$.

Proof. By cases on $e \mid \mu \longrightarrow e' \mid \mu'$.

Case $\boxed{\dfrac{l \notin dom(\mu)}{ref\ v \mid \mu \longrightarrow l \mid [l \mapsto v]\mu}}$:

Since $\Gamma \mid \Sigma \vdash ref\ v : T$, $T = Ref\ T'$ for some T'. Pick some l where $l \notin dom(\mu)$ (and therefore $l \notin dom(\Sigma)$) and let $\Sigma' = [l \mapsto T']\Sigma$. Then $\Gamma \mid \Sigma' \vdash l : Ref\ T'$ and $\Gamma \mid \Sigma' \vdash [l \mapsto v]\mu$.

Towards Type Safety

Case $\boxed{\dfrac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu}}$:

So $\Gamma \mid \Sigma \vdash !l : T$. By inversion, $\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ T$ and then $\Sigma(l) = T$. Then because μ is well typed, we have $\Gamma \mid \Sigma \vdash v : T$.

Case $\boxed{l := v \mid \mu \longrightarrow \mathit{unit} \mid [l \mapsto v]\mu}$:

We have $\Gamma \mid \Sigma \vdash l := v : T$, then by inversion $T = \mathit{Unit}$ and for some T', $\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ T'$ and $\Gamma \mid \Sigma \vdash v : T'$. Then again by inversion, $\Sigma(l) = T'$, and then because μ is well typed we have $\Gamma \mid \Sigma \vdash \mu(l) : T'$. Then because v has the same type as $\mu(l)$, we have $\Gamma \mid \Sigma \vdash [l \mapsto v]\mu$. And of course $\Gamma \mid \Sigma \vdash \mathit{unit} : \mathit{Unit}$.

(Remaining cases are unchanged from the previous proof.)

Exceptions

$$e \quad ::= \quad \ldots \mid \mathit{raise}\ e \mid \mathit{try}\ e\ \mathit{with}\ e$$

Type Rules:

$$\frac{\Gamma \vdash e : T_{exn}}{\Gamma \vdash \mathit{raise}\ e : T} \qquad \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T_{exn} \to T}{\Gamma \vdash \mathit{try}\ e_1\ \mathit{with}\ e_2 : T}$$

There are several possible choices for T_{exn}:

1. Set T_{exn} to something like *Int* or *String*
2. Use an extensible type, such as extensible variants in ML or a base class like `Throwable` in Java.

Exceptions: Operational Semantics

The main computational rules are

$$\mathit{try}\ v\ \mathit{with}\ e \longrightarrow v$$

$$\mathit{try}\ (\mathit{raise}\ v)\ \mathit{with}\ e \longrightarrow e\ v$$

We also need to specify how an exception propagates up. In a small-step semantics with congruence rules, we have the following for propagating through function applications:

$$(\mathit{raise}\ v)\ e \longrightarrow \mathit{raise}\ v$$

$$v_1\ (\mathit{raise}\ v_2) \longrightarrow \mathit{raise}\ v_2$$

You also need similar rules for propagating through **every** other kind of expression.

Exceptions: Propagation with Contexts

We can't add a case for try/with to E because then an exception could skip over a surrounding try/with. Instead we use two kinds of contexts:

$$\begin{array}{lcl} E & ::= & \ldots \mid \mathit{raise}\ E \\ E_h & ::= & [] \mid (E_h\ e) \mid (v\ E_h) \mid \mathit{raise}\ E_h \mid \mathit{try}\ E_h\ \mathit{with}\ e \end{array}$$

$$(1)\ \frac{e \longrightarrow e'}{E_h[e] \longmapsto E_h[e']} \qquad (2)\ \frac{}{E[\mathit{raise}\ v] \longmapsto \mathit{raise}\ v}$$

$$(3)\ \frac{}{E_h[\mathit{try}\ E[\mathit{raise}\ v]\ \mathit{with}\ e] \longmapsto E_h[\mathit{try}\ (\mathit{raise}\ v)\ \mathit{with}\ e]}$$

(2) is needed for uncaught exceptions.

(Source: *Programming Languages and Lambda Calculi* by Felleisen and Flatt)

Normalization

Definition

A program is **normalizable** if evaluation halts after a finite number of steps.

Theorem

Every well typed program of the simply typed lambda calculus is normalizable.

The proof of this theorem demonstrates a proof technique called "logical relations".

Logical Relations

- Motivation: need a stronger induction hypothesis than just "e is normalizable".
- We define for each type T a predicate R_T on programs (closed expressions) of type T.
- The predicate $R_T(e)$ will be stronger then "e is normalizable".
- The definition for R_T we'll use is:

$$\frac{e \text{ is normalizable}}{R_{bool}(e)} \qquad \frac{e \text{ is normalizable} \quad R_{T_1}(e') \longrightarrow R_{T_2}(e\ e')}{R_{T_1 \to T_2}(e)}$$

- The family of relations $\{R_T\}$ for all the types T in the langauge is called a **logical relation**.

Normalization: proof outline

1. Show that every well typed program is in R_T.
2. Show that every program in R_T is normalizable. (This step is immediate from the definition of R_T.)

Every well typed program is in R_T

Lemma

If $\vdash e : T$ *then* $R_T(e)$.

We'll want to prove this by rule induction on $\vdash e : T$, but when we look at subderivations, we'll be considering open expressions with non-empty environments, so the induction hypothesis will not apply. The solution is to generalize the lemma.

Lemma

If $\Gamma \vdash e : T$ *then* $R_T(S(e))$ *where* S *maps every variable* $x \in dom(\Gamma)$ *to a closed value* v *where, letting* $T' = \Gamma(x)$, *we have* $\vdash v : T'$ *and* $R_{T'}(v)$.

Every well typed program is in R_T

Proof.

Case $\boxed{(1)\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}}$:

Because $x : T \in \Gamma$ we have $\vdash S(x) : T$ and $R_T(S(x))$.

Case $\boxed{(2)\dfrac{}{\Gamma \vdash \mathit{true} : \mathit{bool}}}$:

We have $\vdash \mathit{true} : \mathit{bool}$ and $R_{\mathit{bool}}(\mathit{true})$ because true is already a value.

Case $\boxed{(4)\dfrac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.\ e) : T_1 \to T_2}}$:

To show that $R_{T_1 \to T_2}(S(\lambda x : T_1.e))$ we need to show that 1) $S(\lambda x : T_1.e)$ is normalizable and 2) for any e', $R_{T_1}(e')$ implies $R_{T2}(S(\lambda x : T_1.e)\ e')$. We have 1) because a λ is already a value. It remains to address 2). Because $R_{T_1}(e')$, there is a value v such that $e' \longrightarrow^* v$. In order to apply the induction hypothesis on the body e, we need to show that $R_{T_1}(v)$.

Interlude

Lemma

If $\vdash e : T$ *and* $e \longrightarrow e'$, *then* $R_T(e)$ *iff* $R_T(e')$.

Proof.

By induction on T. First, it is clear that e halts iff e' halts.

If $T = \mathit{bool}$ then there is nothing more to show.

If $T = T_1 \to T_2$ then we need to show 1) $R_T(e) \longrightarrow R_T(e')$ and 2) $R_T(e) \longleftarrow R_T(e')$. For 1) suppose $s : T_1$ and $R_{T_1}(s)$. By definition we have $R_{T_2}(e\ s)$. Using the assumption $e \longrightarrow e'$ we have $(e\ s) \longrightarrow (e'\ s)$. Then by the induction hypothesis for T_2 we have $R_{T_2}(e'\ s)$. We therefore have $R_T(e')$. Direction 2) is similar.

Back to: Every well typed program is in R_T

So now we know that $R_{T_1}(v)$ and can apply the induction hypothesis, so $R_{T_2}(S[x \mapsto v](e)\ s)$. But $((\lambda x : T_1.\ S(e))\ e') \longrightarrow^* S[x \mapsto v]e$, and then using the lemma again gives us $R_{T_2}((\lambda x : T_1.S(e))\ e')$. Then by definition, we have $R_{T_1 \to T_2}(S(\lambda x : T_1.\ e))$.

Case $\boxed{(5)\dfrac{\Gamma \vdash e_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash (e_1\ e_2) : T_{12}}}$:

From the induction hypotheses we have $R_{T_{11} \to T_{12}}(S(e_1))$ and $R_{T_{11}}(S(e_2))$. Then by the definiton of $R_{T_{11} \to T_{12}}$ we have $R_{T_{12}}(S(e_1\ e_2))$.