Operational Semantics and Effects

Recall

- Operational Semantics:

$$E[e_0] \mapsto E[e_1] \mapsto E[e_2] \ldots \mapsto v$$

- Stuck terms do not typecheck

- Reductions preserve typability

- Type safety follows

Robust framework

- The above framework is quite robut

- If we change the language by adding a new construct, the old reduction rules and their proofs usually remain valid (almost)

- Let's try adding assignments, exceptions, and continuations.

Assignments

- New syntax:

$$e \ ::= \ \ldots \mid \mathsf{ref}\ e \mid \mathsf{deref}\ e \mid \mathsf{setref}\ e_1\ e_2$$

- Syntactic sugar:

$$\begin{aligned} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 &= (\lambda x.e_2)\ e_1 \\ e_1; e_2 &= (\lambda \_.e_2) e_1 \end{aligned}$$

- Examples:

$$\begin{aligned} inc &= \lambda r.\ \mathsf{setref}\ r\ (\mathsf{add1}\ (\mathsf{deref}\ r)) \\ n &= \texttt{let}\ r = \mathsf{ref}\ 5\ \texttt{in}\ inc\ r; \mathsf{deref}\ r + 2 \\ countadd1\ r &= \lambda x.inc\ r; \mathsf{add1}\ x \end{aligned}$$

Semantics

- Evaluation contexts:

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & \textsf{ref}\ E \mid \textsf{deref}\ E \mid \textsf{setref}\ E\ e \mid \textsf{setref}\ v\ E \end{array}$$

- Locations ℓ

- Stores

$$S\ ::=\ (\ell_1, v_1), \ldots, (\ell_n, v_n)$$

- Expressions and values may contain locations:

$$\begin{array}{rrcl} & e & ::= & \ldots \mid \ell \\ & v & ::= & \ldots \mid \ell \\ (\textit{Programs}) & p & ::= & (S, e) \end{array}$$

Reductions

- Old reductions:

$$
\begin{array}{rcl}
(S, E[\mathsf{add1}\ 0]) & \mapsto & (S, E[1]) \\
(S, E[\mathsf{add1}\ 1]) & \mapsto & (S, E[2]) \\
& \vdots & \\
(S, E[\mathsf{not\ false}]) & \mapsto & (S, E[\mathsf{true}]) \\
(S, E[\mathsf{not\ true}]) & \mapsto & (S, E[\mathsf{false}]) \\
\\
(S, E[(\lambda x.e)v]) & \mapsto & (S, E[e[v/x]])
\end{array}
$$

- New reductions

$$
\begin{array}{rcll}
(S, E[\mathsf{ref}\ v]) & \mapsto & (S \cup (\ell, v), E[\ell]) & \ell \notin S \\
(S, E[\mathsf{deref}\ \ell]) & \mapsto & (S, E[S(\ell)]) & \\
(S \cup (\ell, \_), E[\mathsf{setref}\ \ell\ v]) & \mapsto & (S \cup (\ell, v), E[v]) &
\end{array}
$$

Example

- Given:

$$\begin{aligned} inc &= \lambda r.\ \mathsf{setref}\ r\ (\mathsf{add1}\ (\mathsf{deref}\ r)) \\ n &= \mathtt{let}\ r = \mathsf{ref}\ 5\ \mathtt{in}\ inc\ r; \mathsf{deref}\ r + 2 \end{aligned}$$

- Calculate:

$$\begin{aligned} (\emptyset, n) &\mapsto (\emptyset, \mathtt{let}\ r = \mathsf{ref}\ 5\ \mathtt{in}\ inc\ r; \mathsf{deref}\ r + 2) \\ &\mapsto ((\ell, 5), \mathtt{let}\ r = \ell\ \mathtt{in}\ inc\ r; \mathsf{deref}\ r + 2) \\ &\mapsto ((\ell, 5), inc\ \ell; \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 5), \mathsf{setref}\ \ell\ (\mathsf{add1}\ (\mathsf{deref}\ \ell)); \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 5), \mathsf{setref}\ \ell\ (\mathsf{add1}\ 5); \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 5), \mathsf{setref}\ \ell\ 6; \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 6), 6; \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 6), \mathsf{deref}\ \ell + 2) \\ &\mapsto ((\ell, 6), 6 + 2) \\ &\mapsto ((\ell, 6), 8) \end{aligned}$$

Types

- Syntax of types

$$\tau \ ::= \ \mathsf{int} \mid \mathsf{bool} \mid \cdots \mid \tau \rightarrow \tau$$
$$\mid \ \mathsf{ref}\ \tau$$

- Type rules

$$\frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash \mathsf{ref}\ e\colon \mathsf{ref}\ \tau} \qquad \frac{\Gamma \vdash e\colon \mathsf{ref}\ \tau}{\Gamma \vdash \mathsf{deref}\ e\colon \tau}$$

$$\frac{\Gamma \vdash e_1\colon \mathsf{ref}\ \tau \quad \Gamma \vdash e_2\colon \tau}{\Gamma \vdash \mathsf{setref}\ e_1\ e_2\colon \tau}$$

Subject reduction

- Reductions rewrite stores and expressions
- During evaluation, expressions contain locations
- Need to type stores and locations
- How?

$$\frac{}{\Gamma \vdash \ell \colon ???}$$

Signatures

- Signature Σ maps locations to types

- Type rules

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathsf{ref}\ e : \mathsf{ref}\ \tau} \qquad \frac{\Gamma \vdash_\Sigma e : \mathsf{ref}\ \tau}{\Gamma \vdash_\Sigma \mathsf{deref}\ e : \tau}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathsf{setref}\ e_1\ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash_\Sigma \ell : \mathsf{ref}\ (\Sigma(\ell))}$$

Typing stores

- Store must be consistent with the signature:

$$\frac{S\colon\Sigma \quad \Gamma \vdash_\Sigma e\colon \tau}{\Gamma \vdash_\Sigma (S, e)\colon \tau}$$

- Example:

$$\begin{aligned} S &= (\ell_1, 5), (\ell_2, \mathsf{true}), (\ell_3, (\lambda x.x + (\mathsf{deref}\ \ell_1))) \\ \Sigma &= (\ell_1, \mathsf{int}), (\ell_2, \mathsf{bool}), (\ell_3, \mathsf{int}{\longrightarrow}\mathsf{int}) \end{aligned}$$

- In what order do we compare them?

$$\begin{aligned} S &= (\ell_1, (\lambda x.x + (\mathsf{deref}\ \ell_2\ 0))), (\ell_2, (\lambda x.x + (\mathsf{deref}\ \ell_1\ 0))) \\ \Sigma &= (\ell_1, \mathsf{int}{\longrightarrow}\mathsf{int}), (\ell_2, \mathsf{int}{\longrightarrow}\mathsf{int}) \end{aligned}$$

- Another way to express recursion!

Typing stores

- $S : \Sigma$ if

- the domain of S is equal to the domain of Σ (they talk about the same locations)

- for each location ℓ in the domain of S we have:

$$\vdash_\Sigma S(\ell) : \Sigma(\ell)$$

(Typing relative to the entire signature to allow recursion)

Recursion

- $fc = \mathsf{ref}\ (\lambda x.x)$

  fc: $\mathsf{ref}\ (\mathsf{int} \longrightarrow \mathsf{int})$

- $f = \lambda n.\mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ n * (\mathsf{deref}\ fc\ (n - 1))$

  f: $\mathsf{int} \longrightarrow \mathsf{int}$

- $\mathsf{setref}\ fc\ f$

- Calculate (a bit informally):

$$
\begin{aligned}
f\ 5 &\mapsto \mathtt{if}\ 5 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 5 * (\mathsf{deref}\ fc\ (5 - 1)) \\
&\mapsto 5 * (\mathsf{deref}\ fc\ (5 - 1)) \\
&\mapsto 5 * (f\ (5 - 1)) \\
&\mapsto 5 * (f\ 4) \\
&\mapsto 5 * (\mathtt{if}\ 4 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 4 * (\mathsf{deref}\ fc\ (4 - 1)))
\end{aligned}
$$

Subject reduction

- Perhaps we can prove something like

  If $\Gamma \vdash_{\Sigma} (S, e){:}\,\tau$ and $(S, e) \mapsto (S', e')$ then
  $\Gamma \vdash_{\Sigma} (S', e'){:}\,\tau$

- Counterexample. We have:
  - $\emptyset \vdash_{\emptyset} (\emptyset, \mathsf{ref}\ 5){:}\,\mathsf{ref\ int}$
  - $(\emptyset, \mathsf{ref}\ 5) \mapsto ((\ell, 5), \ell)$

- But $\emptyset \not\vdash_{\emptyset} ((\ell, 5), \ell){:}\,\tau$ for any τ

Monotonicity

- Perhaps we can prove something like

  If $\Gamma \vdash_\Sigma (S, e) \colon \tau$ and $(S, e) \mapsto (S', e')$ then
  there exists Σ' such that $\Gamma \vdash_{\Sigma'} (S', e') \colon \tau$

- Proving the conclusion requires that we prove $S' \colon \Sigma'$
- Looking at the rules, we have either $S' = S$ or $S' = S \cup (\ell, v)$ for some new location ℓ
- We need to prove Σ' agrees with S' on all the old locations and perhaps the new location too.
- We must require that $\Sigma' \supseteq \Sigma$

General picture

- Proof technique is the same as for the pure case
- Cannot exactly reuse previous lemmas; but new proofs are similar
- This is good
- This is also bad

References and Subtyping

- Why subtyping?
- Convenient to have: int $\triangleright$ float
- Necessary for objects, records, etc
- Basic idea: add a type rule

$$\frac{\Gamma \vdash e\colon \tau \quad \tau \triangleright \tau'}{\Gamma \vdash e\colon \tau'}$$

Subtyping relation

- Something on base types:

$$\frac{}{\mathsf{int} \rhd \mathsf{float}}$$

- Sanity rules:

$$\frac{}{\tau \rhd \tau} \qquad \frac{\tau_1 \rhd \tau_2 \quad \tau_2 \rhd \tau_3}{\tau_1 \rhd \tau_3}$$

- Function types (methods in OOP)

$$\frac{\tau_1' \rhd \tau_1 \quad \tau_2 \rhd \tau_2'}{\tau_1 \longrightarrow \tau_2 \rhd \tau_1' \longrightarrow \tau_2'}$$

- A function of type $\mathsf{float} \longrightarrow \mathsf{int}$ can be used anywhere a function of type $\mathsf{int} \longrightarrow \mathsf{float}$ is expected.

What about reference types

- Perhaps?

$$\frac{\tau \triangleright \tau'}{\mathsf{ref}\ \tau \triangleright \mathsf{ref}\ \tau'}$$

- This would allow us to give ref 5 the type ref float.

- But then when we dereference a location of type ref float we might get an int which is not expected.

- Array types in Java give up on typechecking; the above is allowed; and an runtime check is performed.

References and polymorphism

- Why polymorphism?
- Reuse the same code with different types: $\lambda x.x$ can be applied to any type
- Introduce type schemas:

$$\begin{aligned} \sigma &::= \forall \alpha.\sigma \mid \tau \\ \tau &::= \ldots \mid \alpha \end{aligned}$$

- The function $\lambda x.x$ would have type $\forall \alpha.\alpha \rightarrow \alpha$.

Polymorphism

- A type rule to introduce a polymorphic value:

$$\frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash e\colon \forall\alpha.\tau} \qquad \alpha \notin \Gamma$$

- A type rule to use a polymorphic value at any type:

$$\frac{\Gamma \vdash e\colon \forall\alpha.\tau}{\Gamma \vdash e\colon \tau[\tau'/\alpha]}$$

References

- For any α we can prove that $\lambda x.x$ has type $\alpha \rightarrow \alpha$
- We can give ref $(\lambda x.x)$ one of the following two types:
  - $\forall \alpha.\mathsf{ref}\ (\alpha \rightarrow \alpha)$, or
  - $\mathsf{ref}\ (\forall \alpha.(\alpha \rightarrow \alpha))$
- The type $\mathsf{ref}\ (\forall \alpha.(\alpha \rightarrow \alpha))$ contains nested polymorphism which is quite complicated to deal with. Most programming languages restrict this in one way or the other, or completely disallow it.
- The type $\forall \alpha.\mathsf{ref}\ (\alpha \rightarrow \alpha)$ makes the system unsound.

Soundness of references and polymorphism

- Consider

$$\texttt{let}\ fr = \mathsf{ref}\ (\lambda x.x)\ \texttt{in}\ \mathsf{setref}\ fr\ not; \mathsf{deref}\ fr 5$$

- Typechecks but applies *not* to 5.