An alternative semantics

Judgments of the form $E \downarrow V$

• “expression $E$ reduces fully to normal form $V$”
• big-step operational semantics

Can formalize different reduction semantics
E.g., call-by-value reduction:

$$
\frac{\lambda \downarrow (\lambda x : T . E) \quad E_1 \downarrow (\lambda x : T . E) \quad E_2 \downarrow V_2 \quad [v_1] \vdash E \downarrow V}{(E_1 \ E_2) \downarrow V}
$$

Comparison with small-step:

• specifies same result values
• simpler, fewer tedious rules
• closely matches recursive interpreter implementation
• not as nice for proofs, since each step is “bigger”

Yet another alternative semantics

Use explicit environments, not substitution
• closer still to real interpreter

(CBV) environment $\rho$: a sequence of $I \vdash V$ pairs
• records the value of each bound identifier

(Big-step) judgments of the form $\rho \vdash E \downarrow V$

• “in environment $\rho$,
  expression $E$ reduces fully to normal form $V$”

$$
\frac{\rho \vdash I \downarrow V \quad \text{if } I = V \in \rho}{\lambda \downarrow (\lambda x : T . E) \quad \rho \vdash (\lambda x : T . E) \quad \rho \vdash E_1 \downarrow V_1 \quad \rho \vdash E_2 \downarrow V_2 \quad \rho \vdash (E_1 \ E_2) \downarrow V \quad \text{if } I \in \text{dom}(\rho)}
$$

A question

What types should be given to the formals below?

$$(\lambda x : T . x \ x)$$

$$ioop = (\lambda z : T . z \ z) \ (\lambda z : T . z \ z)$$

$$Y = (\lambda \ell : T . \ (\lambda x : T . \ell \ (x \ x)) \ (\lambda x : T . \ell \ (x \ x)))$$

Closures

Values become pairs of lambdas and environments

$V ::= \langle \lambda x : T . E, \rho \rangle$

Revised rules:

$$
\frac{\rho \vdash I \downarrow V \quad \text{if } I = V \in \rho}{\rho \vdash I \downarrow V}
$$

$$
\frac{\rho \vdash (\lambda x : T . E) \downarrow (\lambda x : T . \rho) \quad \rho \vdash E_1 \downarrow (\lambda x : T . E, \rho) \quad \rho \vdash E_2 \downarrow V_2 \quad \rho', \ I = V_1 \vdash E \downarrow V_2}{\rho \vdash (E_1 \ E_2) \downarrow V \quad \text{if } I \in \text{dom}(\rho')}
$$

Comparison with substitution-based semantics:

• specifies “equivalent” result values
• apply environment as substitution to lambda to get same result
• but multiple closures represent same substituted lambda
• very close match to interpreter implementation
• much more complicated $\Rightarrow$ bad for proofs
Amazing fact #5: All simply typed $\lambda$-calculus programs terminate!

Cannot assign types to any program involving self-application
- would require infinite or circular or recursive types

But self-application was used for loop, $Y$, etc.
- cannot write looping or recursive programs
  in simply typed $\lambda$-calculus, at least in this way

Thm (Strong normalization).
Every simply typed $\lambda$-calculus term has a normal form.
- all type-correct programs are guaranteed to terminate!

Simply typed $\lambda$-calculus is not Turing-complete!
- bad for expressiveness in a real PL
- good in restricted domains where we need termination guarantees
  - type checkers
  - OS packet filters
  - ...

Adding explicit recursive values

Make simply typed $\lambda$-calculus more expressive by adding a new primitive to define recursive values: $\text{fix}$

Additional syntax:

$E ::= \ldots | \text{fix } E$

Additional typing rule:

$\frac{\Gamma \vdash E : \tau \rightarrow \tau}{\Gamma \vdash \text{fix } E : \tau}$

Additional (small-step) reduction rule:

$\frac{\text{fix } E \rightarrow E (\text{fix } E)}{\text{fix } E}$

Example of use:

$\text{nat} = (* \rightarrow *) \rightarrow * \rightarrow *$

$\text{fact} = \text{fix } (\lambda \text{fact} : \text{nat} \rightarrow \text{nat}.
\lambda n : \text{nat}. \text{if} (\text{isZero } n) \text{ one}
(\text{mul } n (\text{fact} \ (\text{pred } n))))$

Other extensions

Can design more realistic languages by extending $\lambda$-calculus
Formalize semantics using typing rules and reduction rules

Examples:
- ints
- bools
- let
- records
- tagged unions
- recursive types, e.g. lists
- mutable references

Ints

Additional syntax for types, expressions, and values:

$\tau ::= \ldots | \text{int}$

$E ::= \ldots | 0 | \ldots | E_1 + E_2 | \ldots$

$V ::= \ldots | 0 | \ldots$

Additional typing rules:

$\frac{(\text{numeral})}{\Gamma \vdash k : \text{int}}$ if $k \in \text{Nat}$

$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash (E_1 + E_2) : \text{int}}$ [+]

Additional (big-step) evaluation rules:

$\frac{E_1 \downarrow V_1 \quad E_2 \downarrow V_2}{(E_1 + E_2) \downarrow V}$

$\frac{V \downarrow V}{V \downarrow V}$

$\frac{V_1 \downarrow V_2}{V_1 + V_2 \downarrow V}$

Note: didn’t have to change any existing rules
  to add these new features $\Rightarrow$ they’re orthogonal
**Bools**

Additional syntax for types, expressions, and values:

\[
\begin{align*}
\tau & ::= \ldots | \text{bool} \\
E & ::= \ldots | \text{true} | \text{false} \\
& | \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \\
V & ::= \ldots | \text{true} \text{ | false}
\end{align*}
\]

Additional typing rules:

\[
\begin{align*}
\frac{}{\Gamma \vdash \text{true} : \text{bool}} & \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \\
\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \tau}
\end{align*}
\]

Additional (big-step) evaluation rules:

\[
\begin{align*}
\frac{E_1 \Downarrow \text{true}}{\text{if true}} & \quad \frac{E_2 \Downarrow V_2}{(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) \Downarrow V_2} \\
\frac{E_1 \Downarrow \text{false}}{\text{if false}} & \quad \frac{E_3 \Downarrow V_3}{(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) \Downarrow V_3}
\end{align*}
\]

**Let**

Additional syntax for expressions:

\[
E ::= \ldots | \text{let } I = E_1 \text{ in } E_2
\]

Additional typing rules:

**Records**

Additional syntax for types, expressions, and values:

\[
\begin{align*}
\tau & ::= \ldots | \{ I_1 : \tau_1, \ldots, I_k : \tau_k \} \\
E & ::= \ldots | \{ I_1 = E_1, \ldots, I_k = E_k \} \quad \# \ I \ E \\
V & ::= \ldots | \{ I_1 = V_1, \ldots, I_k = V_k \}
\end{align*}
\]

Additional typing rules:

**Tagged unions**

A tagged union type is a primitive version of an ML datatype:
- a set of labeled alternative types

A value of a tagged union type is one of the labels
- tagging a value of the corresponding alternative type
  - in contrast to records whose values have all of the labeled element types

Example:

\[
\begin{align*}
\text{let } u & = (\text{if } \ldots \text{ then } <A=5> \text{ else } <B=true>) \text{ in} \\
& (* u \text{ has type } <A:\text{int}, B:\text{bool} > *) \\
\text{case } u \text{ of } <A-i> & \Rightarrow \text{printInt } i \\
& | <B-b> \Rightarrow \text{printBool } b
\end{align*}
\]

**Tagged unions**

A tagged union type is a primitive version of an ML datatype:
- a set of labeled alternative types

A value of a tagged union type is one of the labels
- tagging a value of the corresponding alternative type
- in contrast to records whose values have all of the labeled element types

Example:

\[
\begin{align*}
\text{let } u & = (\text{if } \ldots \text{ then } <A=5> \text{ else } <B=true>) \text{ in} \\
& (* u \text{ has type } <A:\text{int}, B:\text{bool} > *) \\
\text{case } u \text{ of } <A-i> & \Rightarrow \text{printInt } i \\
& | <B-b> \Rightarrow \text{printBool } b
\end{align*}
\]
Formalizing tagged unions

Additional syntax for types, expressions, and values:

\[
\begin{align*}
\tau & ::= \ldots | \langle I_1 : \tau_1, \ldots, I_k : \tau_k \rangle \\
E & ::= \ldots | \langle I = E' \rangle \\
V & ::= \ldots | \langle I = V \rangle \\
\end{align*}
\]

Additional typing rules:

Lists

Can use records and tagged unions to define lots of data structures, e.g. (non-polymorphic) lists

\[
\begin{align*}
\text{int\_list} & = \langle \text{Nil}\{}(), \\
& \quad \text{Cons}\{\text{hd}\text{::int, tl\text{::int\_list}}\}\rangle \\
\text{a\_list} & = \langle \text{Cons}\{\text{hd}\text{::1,}} \\
& \quad \text{tl}\text{::Cons}\{\text{hd}\text{::2,} \\
& \quad \text{tl}\text{::Nil\{}\} \} \} \rangle \\
\end{align*}
\]

But something here is bogus!

Recursive types

Previously added support for recursive values (e.g. functions):

\[
\text{fix } E
\]

Now add support for recursive types: \(\mu I. \tau\)

- the same as \(\tau\), except that inside \(\tau\), occurrences of \(I\) mean \(\tau\)

Can correct the definition of \(\text{int\_list}\) type:

\[
\begin{align*}
\text{int\_list} & = \mu T. \langle \text{Nil}\{}(), \\
& \quad \text{Cons}\{\text{hd\text{::int, tl\text{::int\_list}}}\}\rangle \\
\end{align*}
\]

Meaning of recursive type:

- infinite expansion of all recursive references
- but written down in a finite way

An infinitely big type can have finite-sized values

because union includes non-recursive base case

A problem

There are many finite ways to write down an infinite type:

\[
\begin{align*}
\text{int\_list}_0 & = \mu T. \langle \text{Nil}\{}(), \\
& \quad \text{Cons}\{\text{hd\text{::int, tl\text{::int\_list}}}\}\rangle \\
\text{int\_list}_1 & = \langle \text{Nil}\{}(), \\
& \quad \text{Cons}\{\text{hd\text{::int, tl\text{::int\_list}_0}}}\rangle \\
\text{int\_list}_2 & = \langle \text{Nil}\{}(), \\
& \quad \text{Cons}\{\text{hd\text{::int, tl\text{::int\_list}_1}}}\rangle \\
\ldots
\end{align*}
\]

All have the same infinite expansion, so they're all the same

But how's the typechecker to implement type equality checking?

One solution: require explicit operations to convert between different forms, then just use syntactic equality testing

- unfold: \(\mu I. \tau \rightarrow [\mu I. \tau / I]\tau\)
- unfold: \(\text{int\_list}_0 \rightarrow \text{int\_list}_1\)
- fold: \([\mu I. \tau / I]\tau \rightarrow \mu I. \tau\)
- fold: \(\text{int\_list}_1 \rightarrow \text{int\_list}_0\)

ML datatypes wire together a combination of recursive types, fold and unfold operations, and tagged unions in a single mechanism
References and mutable state

Additional syntax for types, expressions, and values:

\[
\begin{align*}
\tau & ::= \ldots \mid \tau \text{ref} \\
E & ::= \ldots \mid \text{ref } E \mid ! E \mid E_1 := E_2 \\
V & ::= \ldots \mid \text{ref } V
\end{align*}
\]

Additional typing rules:

Example

let \( r = \text{ref 1} \) in
let \( x = (r := 2) \) in
! r

Stores and locations

Add an evaluation context to store contents of mutable memory

Location \( l \): a location in mutable memory
- fresh location allocated by \( \text{ref } E \) expression
- locations are values, not \( \text{ref } V \)

Store \( \sigma \): a sequence of \( l \langle V \rangle \) pairs
- represents the contents of each memory location
- initialized by \( \text{ref} \)
- accessed by \( ! \)
- updated by \( := \)

Evaluation of a subexpression now takes an input store and yields a result store to use in later evaluation:

- thread the updated stores through evaluation of all subexpressions
- evaluation order now becomes explicit

Different than environment, which changes when entering nested scopes and is restored when exiting, and which is captured by functions and is restored when they’re called

Revised formalization

Additional syntax for types, expressions, and values:

\[
\begin{align*}
\tau & ::= \ldots \mid \tau \text{ref} \\
E & ::= \ldots \mid \text{ref } E \mid ! E \mid E_1 := E_2 \\
V & ::= \ldots \mid l
\end{align*}
\]

(Typing rules unchanged)

Revised (big-step) evaluation rules:

\[
\begin{align*}
\text{[ref]} & \quad \sigma \vdash E \Downarrow V, \sigma' \quad \text{if } l \notin \text{dom}(\sigma') \\
& \quad \sigma \vdash (\text{ref } E) \Downarrow V, \sigma[l=V] \\
\text{[!]} & \quad \sigma \vdash E \Downarrow l, \sigma' \quad \text{if } l \in V \in \sigma' \\
& \quad \sigma \vdash (! E) \Downarrow V, \sigma' \\
\text{[:=]} & \quad \sigma \vdash E_{1} \Downarrow l, \sigma' \quad \sigma' \vdash E_{2} \Downarrow V, \sigma'' \\
& \quad \sigma \vdash (E_{1} := E_{2}) \Downarrow V, \sigma''[l=V]
\end{align*}
\]

Plus have to revise all earlier rules with threaded stores!
Example again

let r = ref 1 in
let x = (r := 2) in
! r