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:

$$[\lambda] \quad \frac{}{(\lambda I{:}\tau.E) \Downarrow (\lambda I{:}\tau.E)}$$

$$[\text{app}] \quad \frac{E_1 \Downarrow (\lambda I{:}\tau.E) \qquad E_2 \Downarrow V_2 \qquad [V_2/I]E \Downarrow V}{(E_1\ E_2) \Downarrow V}$$

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 ρ: a sequence of *I*=*V* pairs

- records the value of each bound identifier

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

- "in environment ρ,
  expression *E* reduces fully to normal form *V*"

$$[\text{var}] \quad \frac{}{\rho \vdash I \Downarrow V} \quad \text{if } I{=}V \in \rho$$

$$[\lambda] \quad \frac{}{\rho \vdash (\lambda I{:}\tau.E) \Downarrow (\lambda I{:}\tau.E)}$$

$$[\text{app}] \quad \frac{\rho \vdash E_1 \Downarrow (\lambda I{:}\tau.E) \qquad \rho \vdash E_2 \Downarrow V_2 \qquad \rho, I{=}V_2 \vdash E \Downarrow V_2}{\rho \vdash (E_1\ E_2) \Downarrow V} \quad \text{if } I \notin \text{dom}(\rho)$$

Closures

Values become pairs of lambdas and environments
V ::= <λ*I*:τ.*E*, ρ>

Revised rules:

$$[\text{var}] \quad \frac{}{\rho \vdash I \Downarrow V} \quad \text{if } I{=}V \in \rho$$

$$[\lambda] \quad \frac{}{\rho \vdash (\lambda I{:}\tau.E) \Downarrow \langle \lambda I{:}\tau.E, \rho \rangle}$$

$$[\text{app}] \quad \frac{\rho \vdash E_1 \Downarrow \langle \lambda I{:}\tau.E, \rho' \rangle \qquad \rho \vdash E_2 \Downarrow V_2 \qquad \rho', I{=}V_2 \vdash E \Downarrow V_2}{\rho \vdash (E_1\ E_2) \Downarrow V} \quad \text{if } I \notin \text{dom}(\rho')$$

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 ⇒ bad for proofs

A question

What types should be given to the formals below?

```
(λx:?. x x)

loop ≡ ((λz:?. z z) (λz:?. z z))

Y ≡ (λf:?. (λx:?. f (x x)) (λx:?. f (x x)))
```

Amazing fact #5: All simply typed λ-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 λ-calculus, at least in this way

Thm (**Strong normalization**).
Every simply typed λ-calculus term has a normal form.

- all type-correct programs are guaranteed to terminate!

Simply typed λ-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 λ-calculus more expressive by adding a new primitive to define recursive values: `fix`

Additional syntax:

E ::= ... | **fix** *E*

Additional typing rule:

[fix] $$\frac{\Gamma \vdash E:\tau \to \tau}{\Gamma \vdash (\texttt{fix}\ E):\tau}$$

Additional (small-step) reduction rule:

[fix] $$\frac{}{(\texttt{fix}\ E) \to E\ (\texttt{fix}\ E)}$$

Example of use:

nat ≡ (*→*)→*→*

fact ≡ **fix** (λfact:*nat*→*nat*.
  λn:*nat*. *if* (*isZero* n) *one*
        (*mul* n (fact (*pred* n))))

Other extensions

Can design more realistic languages by extending λ-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:

τ ::= ... | **int**
E ::= ... | **0** | ... | E_1 **+** E_2 | ...
V ::= ... | **0** | ...

Additional typing rules:

[numeral] $$\frac{}{\Gamma \vdash k:\text{int}} \quad \text{if } k \in \text{Nat}$$

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

Additional (big-step) evaluation rules:

[val] $$\frac{}{V \Downarrow V}$$

[+] $$\frac{E_1 \Downarrow V_1 \qquad E_2 \Downarrow V_2}{(E_1 + E_2) \Downarrow V} \qquad V = V_1 \hat{+} V_2$$

Note: didn't have to change any existing rules to add these new features ⇒ they're orthogonal

Bools

Additional syntax for types, expressions, and values:

τ ::= ... | **bool**
E ::= ... | **true** | **false**
  | **if** E_1 **then** E_2 **else** E_3
V ::= ... | **true** | **false**

Additional typing rules:

$$[\text{true}]\quad \frac{}{\Gamma \vdash \texttt{true}:\text{bool}} \qquad [\text{false}]\quad \frac{}{\Gamma \vdash \texttt{false}:\text{bool}}$$

$$[\text{if}]\quad \frac{\Gamma \vdash E_1:\text{bool} \qquad \Gamma \vdash E_2:\tau \qquad \Gamma \vdash E_2:\tau}{\Gamma \vdash (\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3):\tau}$$

Additional (big-step) evaluation rules:

$$[\text{if}_{\text{true}}]\quad \frac{E_1 \Downarrow \texttt{true} \qquad E_2 \Downarrow V_2}{(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3) \Downarrow V_2}$$

$$[\text{if}_{\text{false}}]\quad \frac{E_1 \Downarrow \texttt{false} \qquad E_3 \Downarrow V_3}{(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3) \Downarrow V_3}$$

Let

Additional syntax for expressions:

E ::= ... | **let** I **=** E_1 **in** E_2

Additional typing rules:

Additional (big-step) evaluation rules:

Records

Additional syntax for types, expressions, and values:

τ ::= ... | **{**I_1**:**τ_1, ..., I_k**:**τ_k**}**
E ::= ... | **{**I_1**=**E_1, ..., I_k**=**E_k**}** | #I E
V ::= ... | **{**I_1**=**V_1, ..., I_k**=**V_k**}**

Additional typing rules:

Additional (big-step) evaluation 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:

```
let u = (if ... then <A=5> else <B=true>) in
(* u has type <A:int, B:bool> *)
case u of <A=i> => printInt i
        | <B=b> => printBool b
```

Formalizing tagged unions

Additional syntax for types, expressions, and values:

```
τ    ::= ... | <I_1:τ_1,...,I_k:τ_k>
E    ::= ... | <I=E>
        | case E of <I_1=I_1'> => E_1
                  | ...
                  | <I_k=I_k'> => E_k
V    ::= ... | <I=V>
```

Additional typing rules:

Additional (big-step) evaluation rules:

Lists

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

```
int_list ≡ <Nil:{},
            Cons:{hd:int, tl:int_list}>

a_list ≡ <Cons={hd=1,
                tl=<Cons={hd=2,
                          tl=<Nil={}>} >} >
```

But something here is bogus!

Recursive types

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

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

- the same as τ, except that inside τ, occurrences of I mean τ

Can correct the definition of `int_list` type:

```
int_list ≡ μT. <Nil:{},
                Cons:{hd:int, tl:T}>
```

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:

```
int_list0 ≡ μT. <Nil:{},
                 Cons:{hd:int, tl:T}>
int_list1 ≡ <Nil:{},
             Cons:{hd:int, tl:int_list0}>
int_list2 ≡ <Nil:{},
             Cons:{hd:int, tl:int_list1}>
...
```

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: int_list0 → int_list1`
- `fold`: $[\mu I.\tau / I]\tau \rightarrow \mu I . \tau$
  - `fold: int_list1 → int_list0`

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:

τ **::=** ... | τ **ref**
E ::= ... | **ref** E | **!** E | E_1 **:=** E_2
V ::= ... | **ref** V

Additional typing rules:

Additional (big-step) evaluation rules:

Example

```
let r = 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 `ref` *E* expression
- locations are values, not `ref` *V*

Store σ: a sequence of *l*=*V* pairs

- represents the contents of each memory location
- initialized by `ref`
- accessed by `!`
- updated by `:=`

Evaluation of a subexpression now takes an input store *and yields a result store* to use in later evaluation:
$\sigma \vdash E \Downarrow V, \sigma'$

- 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:

τ **::=** ... | τ **ref**
E ::= ... | **ref** E | **!** E | E_1 **:=** E_2
V ::= ... | l

(Typing rules unchanged)

Revised (big-step) evaluation rules:

$$[\text{ref}] \quad \frac{\sigma \vdash E \Downarrow V, \sigma'}{\sigma \vdash (\texttt{ref}\ E) \Downarrow V, \sigma'[l{=}V]} \quad \text{if } l \notin \text{dom}(\sigma')$$

$$[!] \quad \frac{\sigma \vdash E \Downarrow l, \sigma'}{\sigma \vdash (!\ E) \Downarrow V, \sigma'} \quad \text{if } l{=}V \in \sigma'$$

$$[:=] \quad \frac{\sigma \vdash E_1 \Downarrow l, \sigma' \qquad \sigma' \vdash E_2 \Downarrow V, \sigma''}{\sigma \vdash (E_1 := E_2) \Downarrow V, \sigma''[l{=}V]}$$

Plus have to revise all earlier rules with threaded stores!

Example again

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