Going Imperative

Pure vs. Impure Features

- Pure features
  - Functional abstraction/composition
  - Basic types – booleans, numbers
  - Structured types – tuples, records, sums, lists
  - Forms the backbone of most languages
- Impure features
  - Assignment to mutable variables – reference cells, arrays, etc.
  - Input/output of files
  - Non-local transfer of controls – jumps, exception handling, etc.
  - Also called "side effects," - in most practical languages

A Typical Imperative Program

- Factorial of n:

```
int factorial(int n) {
  int x := 1;
  while (n>1) do
          x := x * n;
          n := n -1;
  endwhile;
  return x;
}
```

Imperative Features

- Variable references and assignments
  - x := 1
  - x denotes a memory location (a reference) which stores value 1
- Sequencing

      x := x * n;

      n := n -1

  - A sequence of commands
  - Procedure composition
  - Recall in lambda-calculus: function composition
    - E.g. (\p. p tru) (\b. b v w)
- Loops
  - while (n>1) do …

References and Assignments

- In pure lambda calculus, variable x is mapped to a value, e.g., 1 (or \w.w w) directly.
- In imperative lambda calculus (or lambda with references), we have a variable y whose value is a reference (or pointer/address) to a mutable memory cell which currently stores 1.
  - E.g. y → 0x0000ffff, 0x0000ffff → 1
- To assign another value to y:
  - y := 5
- To dereference y:
  - !y gives the current content 5.
- To create a new reference y (allocation):
  - y = ref 1.

(at this point y is mapped to a new address which contains 1)

Simply-Typed Lambda Calculus with References (Syntax)

e ::=	**Expressions:**
x	variables
| \x: t .e	abstraction
| (e1 e2)	application
| let x = e1 in e2	let expression
| ref e	reference creation
| !e	dereference
| e1 := e2	assignment
| l	store location
| ()	unit (constant)
v ::=	**Values:**
\x:t . e	abstraction value
| l	store location value
| ()	unit value

References (Machine state)

- Extend the Op semantics with "memory store":

  $M ::= . \mid M, l \mapsto v$

  M is a *partial function* from location to values;

  l is a location that indexes into the store M.
- Evaluation rules now have this form:

  $(M, e) \rightarrow (M', e')$
- (M, e) is a "Machine state".
- Define $M[l \mapsto v]$ (update of store):

  $.[l \mapsto v] = l \mapsto v$

  $(M, l' \mapsto v')[l \mapsto v] = M, l \mapsto v$ if $l = l'$

  or $M, l' \mapsto v', l \mapsto v$ if $l \mathrel{!=} l'$

References (Operational Semantics)

$$\frac{(M, e_1) \rightarrow (M', e_1')}{(M, (e_1\ \ e_2)) \rightarrow (M', (e_1'\ \ e_2))}\ \text{(E - App1)}$$

$$\frac{(M, e_2) \rightarrow (M', e_2')}{(M, (v_1\ \ e_2)) \rightarrow (M', (v_1\ e_2'))}\ \text{(E - App2)}$$

$$\frac{}{(M, (\backslash x : t. e_1)\ \ v_2) \rightarrow (M, e_1[v_2/x])}\ \text{(E - AppAbs)}$$

$$\frac{(M, e_1) \rightarrow (M', e_1')}{(M, \text{let } x = e_1 \text{ in } e_2)) \rightarrow (M', \text{let } x = e_1' \text{in } e_2)}\ \text{(E - Let1)}$$

$$\frac{}{(M,\ \ \text{let } x = v_1 \text{ in } e_2)) \rightarrow (M, e_2[v_1/x])}\ \text{(E - Let2)}$$

References (Operational Semantics, Cont'd)

$$\frac{(M, e) \to (M', e')}{(M, \text{ref } e) \to (M', \text{ref } e')} \text{ (E-Ref)}$$

$$\frac{l \notin dom(M)}{(M, \text{ref } v) \to ((M, l \mapsto v), l)} \text{ (E-RefV)}$$

$$\frac{(M, e) \to (M', e')}{(M, !e) \to (M', !e')} \text{ (E-DeRef)}$$

$$\frac{}{(M, !l) \to (M, M(l))} \text{ (E-DeRefLoc)}$$

$$\frac{(M, e_1) \to (M', e_1')}{(M, e_1 := e_2) \to (M', e_1' := e_2)} \text{ (E-Assign1)}$$

$$\frac{(M, e_2) \to (M', e_2')}{(M, v_1 := e_2) \to (M', v_1 := e_2')} \text{ (E-Assign2)}$$

$$\frac{}{(M, l := v) \to (M[l \mapsto v], ())} \text{ (E-Assign)}$$

References (Typing)

- We define the typing relation for memory store as Σ (or Si):

  $\Sigma ::= . \mid \Sigma, l : t$ (t is the type of value stored at l)

- Our new typing judgment:

$$\Sigma; \Gamma \vdash e : t$$

- Types: t ::= .. | unit | t ref

$$\frac{}{\Sigma;\Gamma \mid\!- x : \Gamma(x)} \text{(T-Var)}$$

$$\frac{\Sigma;\Gamma, x : t_1 \mid\!- e : t_2}{\Sigma;\Gamma \mid\!- \lambda x : t_1.e \; : \; t_1 \rightarrow t_2} \text{(T-Abs)}$$

$$\frac{\Sigma;\Gamma \mid\!- e_1 : t_1 \rightarrow t_2 \quad \Sigma;\Gamma \mid\!- e_2 : t_1}{\Sigma;\Gamma \mid\!- e_1 \; e_2 \; : \; t_2} \text{(T-App)}$$

$$\frac{}{\Sigma;\Gamma \mid\!- () : \text{unit}} \text{(T-Unit)}$$

$$\frac{\Sigma(l) = t}{\Sigma;\Gamma \mid\!- l : t \text{ ref}} \text{(T-Loc)}$$

$$\frac{\Sigma;\Gamma \mid\!- e : t}{\Sigma;\Gamma \mid\!- \text{ref } e : t \text{ ref}} \text{(T-Ref)}$$

$$\frac{\Sigma;\Gamma \mid\!- e : t \text{ ref}}{\Sigma;\Gamma \mid\!- !e : t} \text{(T-Deref)}$$

$$\frac{\Sigma;\Gamma \mid\!- e_1 : t \text{ ref} \quad \Sigma;\Gamma \mid\!- e_2 : t}{\Sigma;\Gamma \mid\!- e_1 := e_2 : \text{unit}} \text{(T-Assign)}$$

SEQUENCE

- Assignment returns unit type: doesn't seem to be useful!
- Sequence gives a string of state changes:

$$x := 3;\ y := 2;\ z := 1;\ \ldots$$

- Syntax:

$$e ::= \ldots \mid e_1 \,;\, e_2$$

- Evaluation:

$$\frac{(M, e_1) \to (M', e_1')}{(M, e_1; e_2) \to (M', e_1'; e_2)}\ (\text{E-Seq1}) \qquad \frac{}{(M, (); e) \to (M, e)}\ (\text{E-Seq2})$$

- Typing:

$$\frac{\Sigma;\Gamma \mid\!- e_1 : \text{unit} \qquad \Sigma;\Gamma \mid\!- e_2 : t}{\Sigma;\Gamma \mid\!- e_1; e_2 : t}\ (\text{T-Var})$$

Example Evaluations

Program:

```
let x = ref 3 in
    let y = x in
        x := (!x) +1;
        !y
```

```
(., let x = ref 3 in
    let y = x in
    x:= (!x) + 1;
    y) →
(l 3, let x = l in
          let y = x in
          x := (!x) + 1;
          !y) →
(l 3, let y = l in
          l := (!l) + 1;
          !y) →
(l 3, l := (!l) + 1; !l) →
(l 3, l := 3 + 1; !l) →
(l 3, l := 4; !l) →
(l 4, (); !l) → (l 4, !l) → (l 4, 4)
```

Type Safety

Definition: A store M is well typed under typing context Γ and store typing Σ, written as

$$\Sigma; \Gamma \vdash M,$$

if dom(M)=dom(Σ) and $\Sigma; \Gamma \vdash M(l) : \Sigma(l)$ for all $l \in$ dom(M).

Lemma 1 (weakening). If $\Sigma; \Gamma \vdash e : t$, and $l \notin$ Dom (Σ), then $\Sigma, l : t; \Gamma \vdash e : t$.

Proof: By induction on the derivation of $\Sigma; \Gamma \vdash e: t$

Following says replacing the content of a cell with a new value of appropriate type doesn't change the type of the store.

Lemma 2. If $\Sigma; \Gamma \vdash M$, $\Sigma(l) = t$, $\Sigma; . \vdash v: t$, then $\Sigma; \Gamma \vdash M[l \mapsto v]$.

Proof: Immediate from the above definition of store typing.

Type Safety (Cont'd)

Preservation Theorem. If $\Sigma;\Gamma \vdash e : t$, $\Sigma;\Gamma \vdash M$, and $(M, e) \rightarrow (M', e')$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma';\Gamma \vdash e' : t$, $\Sigma';\Gamma \vdash M'$.

($\Sigma' \supseteq \Sigma$ *means* Σ' *agrees with* Σ *on all the old locations*.)

Proof: Exercise.

Progress Theorem. If e is closed and well-typed (i.e. $\Sigma; . \vdash e : t$ for some Σ and t), then either e is a value or for any store M such that $\Sigma; . \vdash M$, there exists an expression e' and store M', such that $(M, e) \rightarrow (M', e')$.

Proof: Exercise.

While Loop

- Loops are essential in imperative programs:

```
while (!n>1) do
        x := !x * !n;
        n := !n -1
```

- Syntax:

  e::= … | while e1 do e2

- Evaluation:

$$\frac{}{(M, \text{while } e_1 \text{ do } e_2) \rightarrow (M, \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else } ())} \text{(E - While)}$$

- Typing:

$$\frac{\Sigma;\Gamma \mid - e_1 : \text{bool} \quad \Sigma;\Gamma \mid - e_2 : \text{unit}}{\Sigma;\Gamma \mid - \text{while } e_1 \text{ do } e_2 : \text{unit}} \text{(T - While)}$$

Factorial (Imperative Style)

```
let factorial =
  λn.  let m = ref n
         in
           let x = ref 1
           in
             (while (!m >1) do
                    x := !x * !m;
                    m := !m -1);
             !x
  in factorial 10
```

- The above program computes 10!

Exception Handling

- Real world programs need to deal with errors and exceptions.
- When exception happens, we can
  1. Abort the program, or
  2. Transfer control to an exception handler defined in the program
- We will look at this two cases in turn and then refine both mechanisms to allow extra programmer defined data to be passed from exception sites to handlers.

Raising Exception and Abort the Program

- We add a new expression error, which aborts the evaluation of the whole program.
- **Syntax:**

  e ::= … | error (run-time error)

- **Evaluation:**

$$\frac{}{\text{error } e \rightarrow \text{error}} \text{ (E-AppErr1)} \qquad \frac{}{v \text{ error } \rightarrow \text{error}} \text{ (E-AppErr2)}$$

When exceptions happens, evaluation return error itself.

error is only an expression and **not a value** so above two rules don't overlap:

(\x: nat . 0) error → error

We can think of this as "unwinding" application call stack, discarding intermediate computations.

Raising Exception (Typing)

- **Typing:**

$$\frac{}{\Gamma \vdash \text{error} : t} \quad \text{(T-Error)}$$

- t can be any type:
  - (\x:bool . x) error error: bool
  - (\x:bool . x) (error true) error: bool → bool
- This breaks the uniqueness lemma!
  - Solutions: subtyping, or polymorphic types (introduced later)

Handling Exception

- **Syntax:**

  e ::= …

  | try e_1 with e_2 (trap errors)

- **Evaluation:**

$$\frac{}{\text{try v with e} \rightarrow \text{v}} \text{ (E-TryV)}$$

$$\frac{}{\text{try error with e} \rightarrow \text{e}} \text{ (E-TryError)}$$

$$\frac{e_1 \rightarrow e_1'}{\text{try } e_1 \text{ with } e_2 \rightarrow \text{try } e_1' \text{ with } e_2} \text{ (E-Try)}$$

- **Typing:**

$$\frac{\Gamma \mid -e_1 : t \quad \Gamma \mid -e_2 : t}{\Gamma \mid - \text{try } e_1 \text{ with } e_2 : t} \text{ (T-Try)}$$

Raising Exceptions with Values

- It's sometimes useful to pass values from the error site to the handler: e.g.,

  raise RUN_TIME_ERR

  where RUN_TIME_ERR can be a complex structure.

Syntax:

e::= …

| raise e (raise exception)

Evaluation:

$$\frac{}{(\text{raise } v)\ e \rightarrow \text{raise } v} \quad \text{(E-AppRaise1)} \qquad \frac{}{v_1\ (\text{raise } v_2) \rightarrow \text{raise } v_2} \quad \text{(E-AppRaise2)}$$

$$\frac{e \rightarrow e'}{\text{raise } e \rightarrow \text{raise } e'} \quad \text{(E-Raise)} \qquad \frac{}{\text{raise } (\text{raise } v) \rightarrow \text{raise } v} \quad \text{(E-RaiseRaise)}$$

Raising Exceptions with Values (Cont'd)

$$\frac{}{\text{try v with e} \rightarrow \text{v}} \text{ (E-RaiseV)} \qquad \frac{}{\text{try raise v with e} \rightarrow \text{e v}} \text{ (E-TryRaise)}$$

$$\frac{e_1 \rightarrow e_1'}{\text{try } e_1 \text{ with } e_2 \rightarrow \text{try } e_1' \text{ with } e_2} \text{ (E-Try)}$$

- Typing:

$$\frac{\Gamma \mid\!- e : t_{exn}}{\Gamma \mid\!- \text{raise } e : t} \text{ (T-Raise)} \qquad \frac{\Gamma \mid\!- e_1 : t \quad \Gamma \mid\!- e_2 : t_{exn} \rightarrow t}{\Gamma \mid\!- \text{try } e_1 \text{ with } e_2 : t} \text{ (T-Try)}$$

Several Choices of T_{exn}

- t_{exn} = nat:
  - Numeral error code (similar to errno).
  - 0 being success.
  - Need to look up a table for the code.
- t_{exn} = string:
  - Avoids look-up
  - Display a message
  - Handler might have to parse the string

- t_{exn} = <divisionByZero: unit,
  overflow: unit,
  fileNotFound: string,
  …>
  - Labeled Variant type
  - Allow handler to distinguish between different type of exceptions
  - Different except can carry different type of information
  - Inflexible: not programmer-defined

- Extensible variant type: exn (in ML)
- Java Exception Class: using sub-classes
  - Exception extends Throwable
  - Any instance of Exception is a user-defined exception class