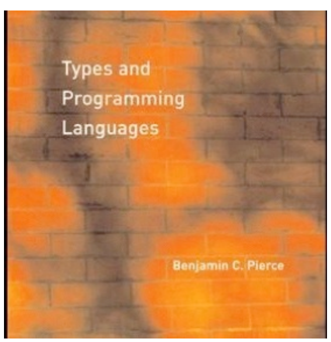
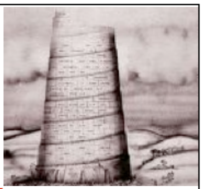
References and Exceptions

CS 565

Lecture 14



References



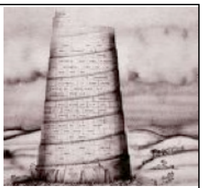
In most languages, variables are mutable:

- it serves as a name for a location
- the contents of the location can be overwritten, and still be referred to by the same name

In ML, variables only name values:

- bindings are immutable
- introduce a new class of values called references.
- A variable bound to mutable location will have type `ref` τ

Basic Operations



Create a reference:

- `ref s`: returns a reference to a location that contains the value denoted by `s`.
- If `s` has type τ, then `ref s :` τ `ref`

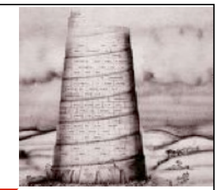
Dereference:

- `!r` : returns the contents of the location referenced by `r`
- If `r` has type τ `ref`, then `!r :` τ

Assignment:

- `r := s`: changes the contents of the location referenced by `r` to hold the value denoted by `s`.
- If `r` has type τ `ref`, and `s` has type τ, then `r := s` has value `unit` of type `Unit`.
- No explicit deallocation operation.

References and Stores



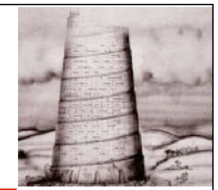
Distinction between a reference and the location pointed to by that reference:

- `r = s`: binds a reference to the location pointed to by `r` to `s`.
- Thus,

```
r = s
s := 13
```

- `r` and `s` are aliases for the same location

References and Shared State



Implement implicit communication channels:

```
c = ref 0
incc = λx:Unit . (c:=succ(!c); !c)
decc = λx:Unit . (c:=pred(!c); !c)
incc unit → 1
decc unit → 0
```

Package both operations together:

```
o = {i = incc,d = decc}
```

We have now have a simple form of object: a collection of operations that share access to common state

References to Complex Types

A location can hold values of any type

Example:

```
newarray = λz:Unit. ref (λn:Nat. 0)
newarray : Unit → (Nat→Nat) ref

lookup = λa:ref(Nat→Nat).λn:Nat.(!a) n;
lookup: (Nat→Nat)ref → Nat → Nat

update = λ a:(Nat → Nat) ref.
  λm:Nat. λv:Nat. let old = !a in
   a:=(λn:Nat.if equal m n then v else old n)
update:(Nat→Nat)ref → Nat → Nat → Unit
```

Typing Rules

$$\frac{\Gamma \vdash t : T\,\mathbf{ref}}{\Gamma \vdash\, !t : T} \quad \textrm{T\_D}\textsc{eref}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbf{ref}\, t : T\,\mathbf{ref}} \quad \textrm{T\_R}\textsc{ef}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : T\,\mathbf{ref} \\ \Gamma \vdash t' : T\end{array}}{\Gamma \vdash t := t' : \mathbf{Unit}} \quad \textrm{T\_A}\textsc{ssign}$$

Evaluation

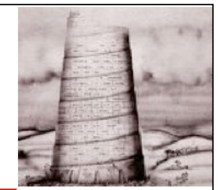
How do we capture the operational (runtime) behavior of reference operations?

- What does it mean to “allocate” storage?
- What does it mean to “assign” to a location?

Think of the store as an array of values

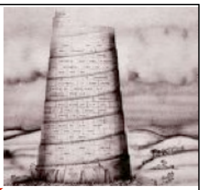
- rather than think of references as addresses (numbers), think of them as elements of a set L of store locations

Evaluation



t	$::=$			terms:
	$\|$	$\mathbf{true}$		constant true
	$\|$	$\mathbf{false}$		constant false
	$\|$	$\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$		conditional
	$\|$	$\mathbf{x}$		variable
	$\|$	$\lambda\,\mathbf{x} : T\,.\,t$		abstraction
	$\|$	$t_1\ t_2$		application
	$\|$	$[\,\mathbf{x} \mapsto v\,]\,t$	M	
	$\|$	nv		integer
	$\|$	$t + t'$		addition
	$\|$	$(\,t\,)$	M	
	$\|$	$!\,t$		dereference
	$\|$	$\mathbf{ref}\ t$		new reference
	$\|$	$t := t'$		assignment
	$\|$	$\mathbf{unit}$		unit
	$\|$	l		address
v	$::=$			values:
	$\|$	$\mathbf{true}$		true value
	$\|$	$\mathbf{false}$		false value
	$\|$	$\lambda\,\mathbf{x} : T\,.\,t$		abstraction value
	$\|$	nv		integer value
	$\|$	$\mathbf{unit}$		unit
	$\|$	l		address

Evaluation



$\boxed{t\,,\,\mu \Downarrow t'\,,\,\mu'}$ Evaluation

$$\frac{}{\lambda \mathbf{x} : T\,.\,t\,,\,\mu \Downarrow \lambda \mathbf{x} : T\,.\,t\,,\,\mu} \quad \text{E\_LAM}$$

$$\frac{\begin{array}{l} t_1\,,\,\mu \Downarrow \lambda \mathbf{x} : T\,.\,t_1'\,,\,\mu' \\ t_2\,,\,\mu' \Downarrow v\,,\,\mu'' \\ [\,\mathbf{x} \mapsto v\,]\,t_1'\,,\,\mu'' \Downarrow v'\,,\,\mu''' \end{array}}{t_1\ t_2\,,\,\mu \Downarrow v'\,,\,\mu'''} \quad \text{E\_APP}$$

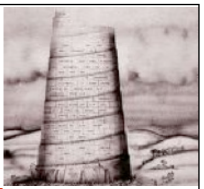
$$\frac{\begin{array}{c} t_1\,,\,\mu \Downarrow \mathbf{true}\,,\,\mu' \\ t_2\,,\,\mu' \Downarrow v\,,\,\mu'' \end{array}}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\,,\,\mu \Downarrow v\,,\,\mu''} \quad \text{E\_IFT}$$

$$\frac{\begin{array}{c} t_1\,,\,\mu \Downarrow \mathbf{false}\,,\,\mu' \\ t_3\,,\,\mu' \Downarrow v\,,\,\mu'' \end{array}}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\,,\,\mu \Downarrow v\,,\,\mu''} \quad \text{E\_IFF}$$

$$\frac{\begin{array}{c} t_1\,,\,\mu \Downarrow nv_1\,,\,\mu' \\ t_2\,,\,\mu' \Downarrow nv_2\,,\,\mu'' \\ nv_1 + nv_2 = nv_3 \end{array}}{t_1 + t_2\,,\,\mu \Downarrow nv_3\,,\,\mu''} \quad \text{E\_PLUS}$$

$$\frac{}{v\,,\,\mu \Downarrow v\,,\,\mu} \quad \text{E\_VAL}$$

Evaluation



$$\frac{\mu'(l) = v}{!\,t\,,\,\mu \Downarrow v\,,\,\mu'} \quad \text{E\_DEREF}$$

$$\frac{\begin{array}{l} t\,,\,\mu \Downarrow l\,,\,\mu' \\ t'\,,\,\mu' \Downarrow v\,,\,\mu'' \\ \mu''' = \mu''[\,l \mapsto v\,] \end{array}}{t := t'\,,\,\mu \Downarrow \mathbf{unit}\,,\,\mu'''} \quad \text{E\_ASSIGN}$$

$$\frac{\begin{array}{l} t\,,\,\mu \Downarrow v\,,\,\mu' \\ \mu'' = \mu'[\,l \mapsto v\,] \\ l \notin \mathbf{dom}(\mu') \end{array}}{\mathbf{ref}\;t\,,\,\mu \Downarrow l\,,\,\mu''} \quad \text{E\_ALLOC}$$

Locations

Extend typing rule to accommodate locations:

$$\frac{\Gamma, \mu \vdash \mu(l) : \tau}{\Gamma, \mu \vdash l : \text{Ref } \tau}$$

The type of a location depends upon the contents of the store:

- If μ=`[l₁ ↦ Unit,l₂ ↦ Unit]`, then l_2 has type `Unit`
- If μ=`[l₁ ↦ Unit,l₂ ↦ Unit→Unit]`, then l_2 has type `Unit→Unit`

Problem

This type rule isn't very satisfactory

- large type derivations
- doesn't handle cycles in the store

Suppose the store is defined by:

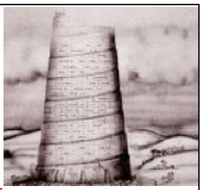
$$
\begin{aligned}
[\; & \mathtt{l}_1 \mapsto \lambda\mathtt{x{:}Nat.99}, \\
& \mathtt{l}_2 \mapsto \lambda\mathtt{x{:}Nat.(!l}_1\mathtt{)\ x}, \\
& \mathtt{l}_3 \mapsto \lambda\mathtt{x{:}Nat.(!l}_2\mathtt{)\ x}, \qquad \ldots \;]
\end{aligned}
$$

Now, typing ln requires calculating types of $\mathtt{l}_1, \ldots, \mathtt{l}_{n-1}$

Suppose we have:

$$[\mathtt{l}_1 \mapsto \lambda\mathtt{x{:}Nat.(!l}_2\mathtt{)\ x},\ \mathtt{l}_2 \mapsto \lambda\mathtt{x{:}Nat.(!l}_1\mathtt{)x}]$$

Issues

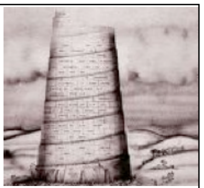


How do we create cycles?

```
let r1 = ref (λx:Nat. 0)
    r2 = ref (λx:Nat.(!r1) x)
in  (r1 := λx:Nat.(!r2) x;
     r2)
```

Unnecessary for us to recalculate the type of a location every time it is mentioned:

- we know its type at the point it is declared
- all other values stored in the location must share that type



Store Typings

For a store that contains:

$$[\ \mathtt{l}_1 \mapsto \lambda\mathtt{x}\!:\!\mathtt{Nat.99},\qquad \mathtt{l}_2 \mapsto \lambda\mathtt{x}\!:\!\mathtt{Nat.(!l}_1\mathtt{)\ x},$$
$$\mathtt{l}_3 \mapsto \lambda\mathtt{x}\!:\!\mathtt{Nat.(!l}_2\mathtt{)\ x},\ \ldots\ ]$$

a reasonable typing would be:

$$[\ \mathtt{l}_1 \mapsto \mathtt{Nat}\to\mathtt{Nat},\ \mathtt{l}_2 \mapsto \mathtt{Nat}\to\mathtt{Nat},$$
$$\mathtt{l}_3 \mapsto \mathtt{Nat}\to\mathtt{Nat},\ \ldots\ ]$$

Store Typings

A store typing Σ describes the store μ in which we intend to evaluate term `t`. We use Σ to lookup the types of locations referenced in `t`.

$$\frac{\Sigma(l) = T}{\Gamma, \Sigma \vdash l : T\,\mathbf{ref}} \quad \textsc{T\_Address}$$

Need to propagate Σ to all the other type rules defined earlier.

Store Typings

A given store may have multiple store typings:

Suppose

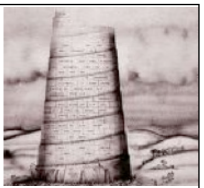
$$\mu = [\ \texttt{l} \mapsto \lambda\texttt{x:Unit. (!l) x}]$$

Then,

$$\Sigma_1 = \texttt{l} \mapsto \texttt{Unit} \rightarrow \texttt{Unit}$$

$$\Sigma_2 = \texttt{l} \mapsto \texttt{Unit} \rightarrow \texttt{Unit} \rightarrow \texttt{Unit}$$

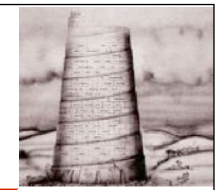
Exceptions



An exception is a construct that allows programmers deal with exceptional conditions (e.g., errors)

- exception handler: code that is associated with an exception that is invoked when an exception is raised.
- raising an exception causes the computation to transfer control to the closest enclosing handler (in the dynamic context).

First step: Errors



An error is a special term that when evaluated stops evaluation of the term.

- Values: $v ::= n \mid \mathtt{true} \mid \mathtt{false} \mid \lambda\, x{:}\ \tau.\ e \mid$
- Terms: $t ::= x \mid \lambda\, x{:}\tau.t \mid e\ e \mid \mathtt{error}$
- Evaluation rules (Contextual)

  $E ::= [\,] \mid E\ t \mid (\lambda\, x{:}\tau.\ t)\ E$

- Reduction

  $\mathtt{error}\ t \rightarrow \mathtt{error}$

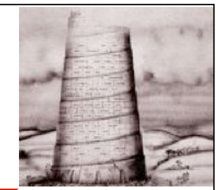
  $v\ \mathtt{error} \rightarrow \mathtt{error}$

- Typing

  $\Gamma \vdash \mathtt{error} : T$ (an error can be of any type)

- What difficulties do we face in expressing error using a big-step semantics?

Typing



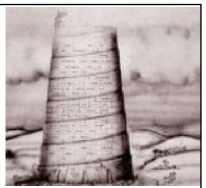
Since error can be of any type, it breaks uniqueness property of types:

- subtyping: allow error to be “promoted” to any other type as necessary by defining it the “minimal” type
- polymorphism: give error the polymorphic type `∀X.X` that can be “instantiated” to any other type as necessary

Why not just use annotations? Consider:

```
(λ x:Nat. x) ((λ y:Bool.13) (error as Bool))
```

Exceptions

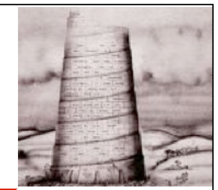


Evaluating error “unwinds” the call-stack until all frames have been discarded, and evaluation returns to the top-level.

Generalizing to exceptions, allows handlers to be inserted between activation frames in the call-stack

- control reverts to the handler that handles the exception raised
- use the first matching handler

Error Handling



Values: `v ::= n | true | false |` λ `x:` τ`. e |`

Terms: `e ::= x |` λ `x:`τ`.e | e e | error |`

`try e with e`

Contexts and Reduction Rules:

`E ::= ... | try E with e`

`r ::= ... | try error with e | try v with e`

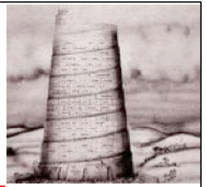
`try v with e` $\rightarrow$ `v`

`try error with e` $\rightarrow$ `e`

Type rule:

$\Gamma \vdash$ `try t with t'`$:\tau$ iff $\Gamma \vdash$ `t`$:\tau$ and $\Gamma \vdash$ `t'`$:\tau$

Exception-Carrying Values

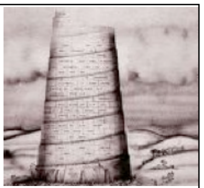


Suppose we want to send information to a handler about the unusual event that triggered the exception

Allow exceptions to carry values

When an exception is raised, supply a value that is an argument to the handler.

Evaluation Rules



Values: `v ::= n | true | false | λ x:T.t`

Terms:

```
t ::= x | λ x:T.t | t t | try t with t |raise t
```

Evaluation contexts and Reduction Rules:

```
E ::= ... | try E with t | raise E
```

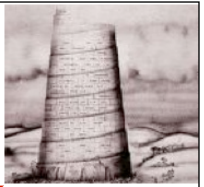
```
try v with t → v
try (raise v) with t → t v
(raise v) t → raise v
v (raise v1) → raise v1
raise (raise v) → raise v
```

Typing Rules

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \text{raise } e : \tau}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e_h : \tau_{exn} \rightarrow \tau}{\Gamma \vdash \text{try } e \text{ with } e_h : \tau}$$

Exception Types



What type should τ_{exn} be?

- Take it to be `Nat`. Corresponds to `errno` convention in Unix.
- Take it to be `String`.
- Take it to be a variant type:

```
τexn = divideByZero: Unit + overflow : Unit +
       fileNotFound : String + ...
```

  Not particularly flexible

Assume τ_{exn} is an extensible variant:

- In ML, there is a single extensible variant type called `exn`.
- exception `l` of `T` means “make sure `l` is different from any other tag present in the variant type τ_{exn}”

  τ_{exn} is henceforth τ_{exn}`+l:T`

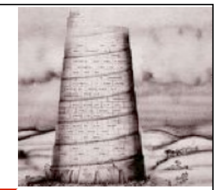
Continuations

Exceptions and errors are instantiations of a more general control feature that allows non-local transfer of control from point in the program to another.

- structured jumps or gotos

Can we generalize (or reify) this notion into our core language?

- result is a continuation: a reified representation (in the form of an abstraction) of a program's control-stack.



Continuations

Define a new primitive `call/cc`:

- Takes as its argument a procedure

```
(lambda (k) e)
```

  and binds to `k` a reified representation of the call-stack at the point of evaluation.
- Can transfer control to this point via application.

Examples

```
call/cc (λ k. (k 3) + 2) + 1  →  4
```

```
val r = ref (λ v. 0)
call/cc (λ k. (r := k;  (k 3) + 2)) + 1     →  4
(!r 4)   →  5
```

```
let f = call/cc (λ k. λ x. k (λ y. x + y))
in f 6  →
12
```

Evaluation and Typing Rules

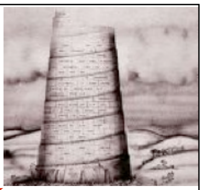
First, consider the evaluation rule in an untyped setting:

```
E[call/cc e] = E[e (λ v. abort (E[v]))]
```

where abort represents the “initial” continuation.

Typing is a bit harder because continuations bound by call/cc can be invoked in several different contexts

An Example in ML



```
1 + call/cc (fn k => hd (if b
                             then [ (k 3) + 1]
                             else 5 :: (k 4)))
```

`k` is invoked in two contexts:

- one expects an integer
- other expects a list

Since continuations never return, how do we choose the result type?

One possible type: $(\tau\ \texttt{cont} \rightarrow \tau) \rightarrow \tau$

- Will revisit this issue when we consider polymorphism.