CIS 500

Software Foundations

Fall 2005

2 November

Recursion

Recursion in $\lambda_{\rightarrow}$

- In $\lambda_{\rightarrow}$, all programs terminate. (Cf. Chapter 12.)
- Hence, untyped terms like `omega` and `fix` are not typable.
- But we can extend the system with a (typed) fixed-point operator...

Example

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));

iseven = fix ff;

iseven 7;
```

New syntactic forms

`t ::=` ...		*terms*
	`fix t`	*fixed point of* `t`

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\begin{array}{c}\texttt{fix}\ (\lambda x{:}T_1.t_2)\\ \longrightarrow [x \mapsto (\texttt{fix}\ (\lambda x{:}T_1.t_2))]t_2\end{array} \qquad \text{(E-FixBeta)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'} \qquad \text{(E-Fix)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \texttt{fix}\ t_1 : T_1} \qquad \text{(T-Fix)}$$

A more convenient form

$$\texttt{letrec}\ x{:}T_1{=}t_1\ \texttt{in}\ t_2 \overset{\text{def}}{=} \texttt{let}\ x = \texttt{fix}\ (\lambda x{:}T_1.t_1)\ \texttt{in}\ t_2$$

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```

References

Mutability

- In most programming languages, variables are mutable — i.e., a variable provides both
  - a name that refers to a previously calculated value, and
  - the possibility of overwriting this value with another (which will be referred to by the same name)
- In some languages (e.g., OCaml), these two features are kept separate
  - variables are only for naming — the binding between a variable and its value is immutable
  - introduce a new class of mutable values (called reference cells or references)
  - at any given moment, a reference holds a value (and can be dereferenced to obtain this value)
  - a new value may be assigned to a reference

Basic Examples

```
let r = ref 5
!r
r := 7
(r:=succ(!r); !r)
(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r)
```

Basic Examples

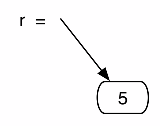
```
let r = ref 5
!r
r := 7
(r:=succ(!r); !r)
(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r)
```

i.e.,

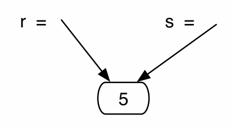
```
((((r:=succ(!r); r:=succ(!r)); r:=succ(!r)); r:=succ(!r)); !r)
```

Aliasing

A value of type `Ref T` is a pointer to a cell holding a value of type `T`.



If this value is “copied” by assigning it to another variable, the cell pointed to is not copied.



So we can change `r` by assigning to `s`:

```
(s:=6; !r)
```

Aliasing all around us

Reference cells are not the only language feature that introduces the possibility of aliasing.

- arrays
- communication channels
- I/O devices (disks, etc.)

The difficulties of aliasing

The possibility of aliasing invalidates all sorts of useful forms of reasoning about programs, both by programmers...

> The function
>
> ```
> λr:Ref Nat. λs:Ref Nat. (r:=2; s:=3; !r)
> ```
>
> always returns `2` unless `r` and `s` are aliases for the same cell.

...and by compilers:

> Code motion out of loops, common subexpression elimination, allocation of variables to registers, and detection of uninitialized variables all depend upon the compiler knowing which objects a load or a store operation could reference.

High-performance compilers spend significant energy on alias analysis to try to establish when different variables cannot possibly refer to the same storage.

The benefits of aliasing

The problems of aliasing have led some language designers simply to disallow it (e.g., Haskell).

But there are good reasons why most languages do provide constructs involving aliasing:

- efficiency (e.g., arrays)
- “action at a distance” (e.g., symbol tables)
- shared resources (e.g., locks) in concurrent systems
- etc.

Example

```
c = ref 0

incc = λx:Unit. (c := succ (!c); !c)

decc = λx:Unit. (c := pred (!c); !c)

incc unit

decc unit

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

```
let newcounter =
  λ_:Unit.
    let c = ref 0 in
    let incc = λx:Unit. (c := succ (!c); !c) in
    let decc = λx:Unit. (c := pred (!c); !c) in
    let o = {i = incc, d = decc} in
    o
```

Syntax

`t` ::=		*terms*
	`unit`	*unit constant*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`ref t`	*reference creation*
	`!t`	*dereference*
	`t:=t`	*assignment*

... plus other familiar types, in examples.

Typing Rules

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1}{\Gamma \vdash \mathtt{!t}_1 : \mathtt{T}_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

Another example

```
BoolArray = Ref (Nat→Bool);

newarray = λ_:Unit. ref (λn:Nat.false);
         : Unit → BoolArray

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

update = λa:BoolArray. λm:Nat. λv:Bool.
           let oldf = !a in
           a := (λn:Nat. if equal m n then v else oldf n);
       : BoolArray → Nat → Bool → Unit
```

Evaluation

What is the value of the expression `ref 0`?

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Specifically, evaluating `ref 0` should allocate some storage and yield a reference (or pointer) to that storage.

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Specifically, evaluating `ref 0` should allocate some storage and yield a reference (or pointer) to that storage.

So what is a reference?

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
- More abstractly: an array of values

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
- More abstractly: an array of values
- Even more abstractly: a partial function from locations to values.

Locations

Syntax of values:

`v`	::=	*values*
	`unit`	*unit constant*
	`λx:T.t`	*abstraction value*
	`l`	*store location*

... and since all values are terms...

Syntax of Terms

`t`	::=	*terms*
	`unit`	*unit constant*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`ref t`	*reference creation*
	`!t`	*dereference*
	`t:=t`	*assignment*
	`l`	*store location*

Aside

Does this mean we are going to allow programmers to write explicit locations in their programs??

No: This is just a modeling trick. We are enriching the "source language" to include some run-time structures, so that we can continue to formalize evaluation as a relation between source terms.

Aside: If we formalize evaluation in the big-step style, then we can add locations to the set of values (results of evaluation) without adding them to the set of terms.

Evaluation

The result of evaluating a term now depends on the store in which it is evaluated. Moreover, the result of evaluating a term is not just a value — we must also keep track of the changes that get made to the store.

I.e., the evaluation relation should now map a term and a store to a reduced term and a new store.

$$\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$$

We use the metavariable μ to range over stores.

Evaluation

An assignment `t₁:=t₂` first evaluates `t₁` and `t₂` until they become values...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\mathtt{:=}\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\mathtt{:=}\mathtt{t}_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\mathtt{:=}\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\mathtt{:=}\mathtt{t}_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

... and then returns `unit` and updates the store:

$$\mathtt{l}\mathtt{:=}\mathtt{v}_2 \mid \mu \longrightarrow \mathtt{unit} \mid [\mathtt{l} \mapsto \mathtt{v}_2]\mu \qquad \text{(E-Assign)}$$

A term of the form `ref t₁` first evaluates inside `t₁` until it becomes a value...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{ref}\ \mathtt{t}_1 \mid \mu \longrightarrow \mathtt{ref}\ \mathtt{t}_1' \mid \mu'} \qquad \text{(E-Ref)}$$

... and then chooses (allocates) a fresh location $\mathtt{l}$, augments the store with a binding from $\mathtt{l}$ to $\mathtt{v}_1$, and returns $\mathtt{l}$:

$$\frac{\mathtt{l} \notin \mathit{dom}(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow \mathtt{l} \mid (\mu, \mathtt{l} \mapsto \mathtt{v}_1)} \qquad \text{(E-RefV)}$$

A term `!t₁` first evaluates in `t₁` until it becomes a value...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{!t}_1 \mid \mu \longrightarrow \mathtt{!t}_1' \mid \mu'} \qquad \text{(E-Deref)}$$

... and then looks up this value (which must be a location, if the original term was well typed) and returns its contents in the current store:

$$\frac{\mu(\mathtt{l}) = \mathtt{v}}{\mathtt{!l} \mid \mu \longrightarrow \mathtt{v} \mid \mu} \qquad \text{(E-DerefLoc)}$$

Evaluation rules for function abstraction and application are augmented with stores, but don't do anything with them directly.

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\ \mathtt{t}_2' \mid \mu'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x}\mathtt{:}\mathtt{T}_{11}\mathtt{.}\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

Aside: garbage collection

Note that we are not modeling garbage collection — the store just grows without bound.

Aside: pointer arithmetic

We can't do any!

Store Typings

Typing Locations

Q: What is the type of a location?

Typing Locations

Q: What is the type of a location?

A: It depends on the store!

E.g., in the store $(l_1 \mapsto \texttt{unit}, l_2 \mapsto \texttt{unit})$, the term $!l_2$ has type `Unit`.

But in the store $(l_1 \mapsto \texttt{unit}, l_2 \mapsto \lambda\texttt{x:Unit.x})$, the term $!l_2$ has type $\texttt{Unit}\to\texttt{Unit}$.

Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \texttt{Ref } T_1}$$

Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \texttt{Ref } T_1}$$

More precisely:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \texttt{Ref } T_1}$$

I.e., typing is now a four-place relation (between contexts, stores, terms, and types).

Problem

However, this rule is not completely satisfactory. For one thing, it can make typing derivations very large!

E.g., if

$$\begin{aligned}(\mu = l_1 &\mapsto \lambda\texttt{x:Nat. 999},\\ l_2 &\mapsto \lambda\texttt{x:Nat. } !l_1 \ (!l_1 \ \texttt{x}),\\ l_3 &\mapsto \lambda\texttt{x:Nat. } !l_2 \ (!l_2 \ \texttt{x}),\\ l_4 &\mapsto \lambda\texttt{x:Nat. } !l_3 \ (!l_3 \ \texttt{x}),\\ l_5 &\mapsto \lambda\texttt{x:Nat. } !l_4 \ (!l_4 \ \texttt{x})),\end{aligned}$$

then how big is the typing derivation for $!l_5$?

Problem!

But wait... it gets worse. Suppose

$$\begin{aligned}(\mu = \; & l_1 \mapsto \lambda\texttt{x:Nat. !}l_2\texttt{ x},\\ & l_2 \mapsto \lambda\texttt{x:Nat. !}l_1\texttt{ x}),\end{aligned}$$

Now how big is the typing derivation for $\texttt{!}l_2$?

Store Typings

Observation: The typing rules we have chosen for references guarantee that a given location in the store is always used to hold values of the same type.

These intended types can be collected into a store typing — a partial function from locations to types.

E.g., for

$$\begin{aligned}\mu = (\; & l_1 \mapsto \lambda\texttt{x:Nat. 999},\\ & l_2 \mapsto \lambda\texttt{x:Nat. !}l_1\texttt{ (!}l_1\texttt{ x)},\\ & l_3 \mapsto \lambda\texttt{x:Nat. !}l_2\texttt{ (!}l_2\texttt{ x)},\\ & l_4 \mapsto \lambda\texttt{x:Nat. !}l_3\texttt{ (!}l_3\texttt{ x)},\\ & l_5 \mapsto \lambda\texttt{x:Nat. !}l_4\texttt{ (!}l_4\texttt{ x)}),\end{aligned}$$

A reasonable store typing would be

$$\begin{aligned}\Sigma = (\; & l_1 \mapsto \texttt{Nat}\to\texttt{Nat},\\ & l_2 \mapsto \texttt{Nat}\to\texttt{Nat},\\ & l_3 \mapsto \texttt{Nat}\to\texttt{Nat},\\ & l_4 \mapsto \texttt{Nat}\to\texttt{Nat},\\ & l_5 \mapsto \texttt{Nat}\to\texttt{Nat})\end{aligned}$$

Now, suppose we are given a store typing Σ describing the store μ in which we intend to evaluate some term `t`. Then we can use Σ to look up the types of locations in `t` instead of calculating them from the values in μ.

$$\frac{\Sigma(l) = \texttt{T}_1}{\Gamma \mid \Sigma \vdash l : \texttt{Ref T}_1} \qquad \text{(T-Loc)}$$

I.e., typing is now a four-place relation between between contexts, store typings, terms, and types.

Final typing rules

$$\frac{\Sigma(l) = \mathtt{T}_1}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref}\ \mathtt{T}_1} \quad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \quad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \quad \text{(T-Assign)}$$

Q: Where do these store typings come from?

Q: Where do these store typings come from?

A: When we first typecheck a program, there will be no explicit locations, so we can use an empty store typing.

So, when a new location is created during evaluation,

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathtt{v}_1)} \quad \text{(E-RefV)}$$

we can observe the type of $\mathtt{v}_1$ and extend the “current store typing” appropriately.

Safety

[on board]