CIS 500

Software Foundations

Fall 2003

22 October

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

We choose OCaml's style, which is easier to work with formally.

So a variable of type `T` in most languages (except OCaml) will correspond to a `Ref T` (actually, a `Ref(Option T)`) here.

Basic Examples

```
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

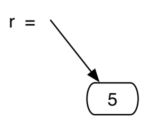
```
r = ref 5
!r
r := 7
(r:=succ(!r); !r)
(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r)
```

i.e.,

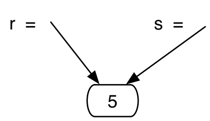
```
((((r:=succ(!r); r:=succ(!r)); r:=succ(!r)); r:=succ(!r)); !r)
```


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

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \vdash \mathtt{ref\ t_1} : \mathtt{Ref\ T_1}} \quad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{Ref\ T_1}}{\Gamma \vdash \mathtt{!t_1} : \mathtt{T_1}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{Ref\ T_1} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_1}}{\Gamma \vdash \mathtt{t_1 := t_2} : \mathtt{Unit}} \quad \text{(T-Assign)}$$

Final example

```
NatArray = Ref (Nat→Nat);

newarray = λ_:Unit. ref (λn:Nat.0);
         : Unit → NatArray

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

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