CS4450: Principles of Programming Languages

Imperative features; reference types

Dr William Harrison

Today: Imperative programming

- Imperative programming
  - i.e., programming with `;` `:=`
  - Case Study:
    - imperative programming in ML
      - “references” = “assignable variables”
      - “reference types”: assign-ability tracked in the type system
- How to interpret “:=” and “;”

Reference types

- ML has types for “assignable variables” called reference types
  - i.e., in order for a variable to be on the l.h.s. of a “:=”, it must have a reference type
  - References are like pointers, but **type-safe**

Example in Standard ML

Create a reference with "**ref**"

```
- val x = ref 1;
val x = ref 1 : int ref
```

reference type

Read a reference with "**!**"

```
- !x;
val it = 1 : int
```

Write a reference with "**!**"

```
- x := !x + 1;
val it = () : unit
```

must use "!" to read and ":=" to write a reference

Sequencing $(e_1 ; \ldots ; e_n)$

- The arguments to a sequence can be anything

```
(1 ; "hey" ; 3.14);
```

- …and the type of the whole sequence is the type of the last thing:

```
val it = 3.14 : real
```

Reference is like a pointer (`int *x`), **except…**

There is no explicit allocation/deallocation of memory*

ML

```
val x = ref 1;
val x = ref 1 : int ref
```

C

```
x = malloc(sizeof int);
```

No casting (i.e., references are type-safe)

C

```
y = (real) *x
```

** …and no possibility of dereferencing a null pointer!*

But there is “aliasing”

```
- val p = ref 9;
val p = ref 9 : int ref
- val q = p;
val q = ref 9 : int ref
- !p;
val it = 9 : int
- !q;
val it = 9 : int
```

```
- p := 5;
val it = () : unit
- !p;
val it = 5 : int
- !q;
val it = 5 : int
```

What is a "side effect"?

Heretofore, the **entire** meaning of a program is its **value**

"`(\ x -> x + x) 2`" is `4`

With imperative features, values tell only part of the story

"`( loc := 2 ; loc := !loc + !loc ; !loc )`" has value `4`

…but this expression also involves hidden (aka, "side") effects

Referential transparency

same expression

```
- val f = (fn y => y + y);
val f = fn : int -> int
- val arg = 2;
val arg = 2 : int
- f arg;
val it = 4 : int
- f arg;
val it = 4 : int
- f arg;
val it = 4 : int
- f arg;
val it = 4 : int
```

same result, no matter how many times it's eval'd

* "fn x =>" in ML is the same as "\ x ->" in Haskell

Side effects negate referential transparency

```
- val x = ref 10;
val x = ref 10 : int ref
- f (x := !x * !x ; !x );
val it = 200 : int
- f (x := !x * !x ; !x );
val it = 20000 : int
- f (x := !x * !x ; !x );
val it = 200000000 : int
- f (x := !x * !x ; !x );

uncaught exception overflow
```

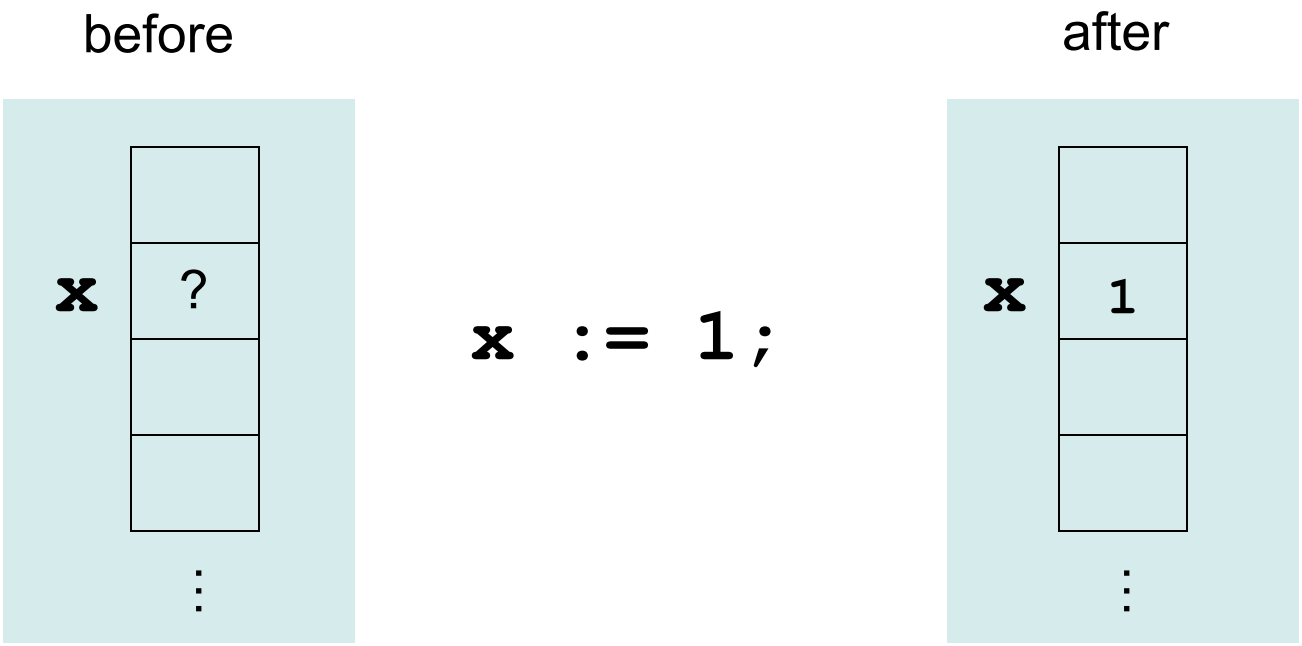
What is the type of "`x:=e`"?

What is the **`Value`** of an assignment?

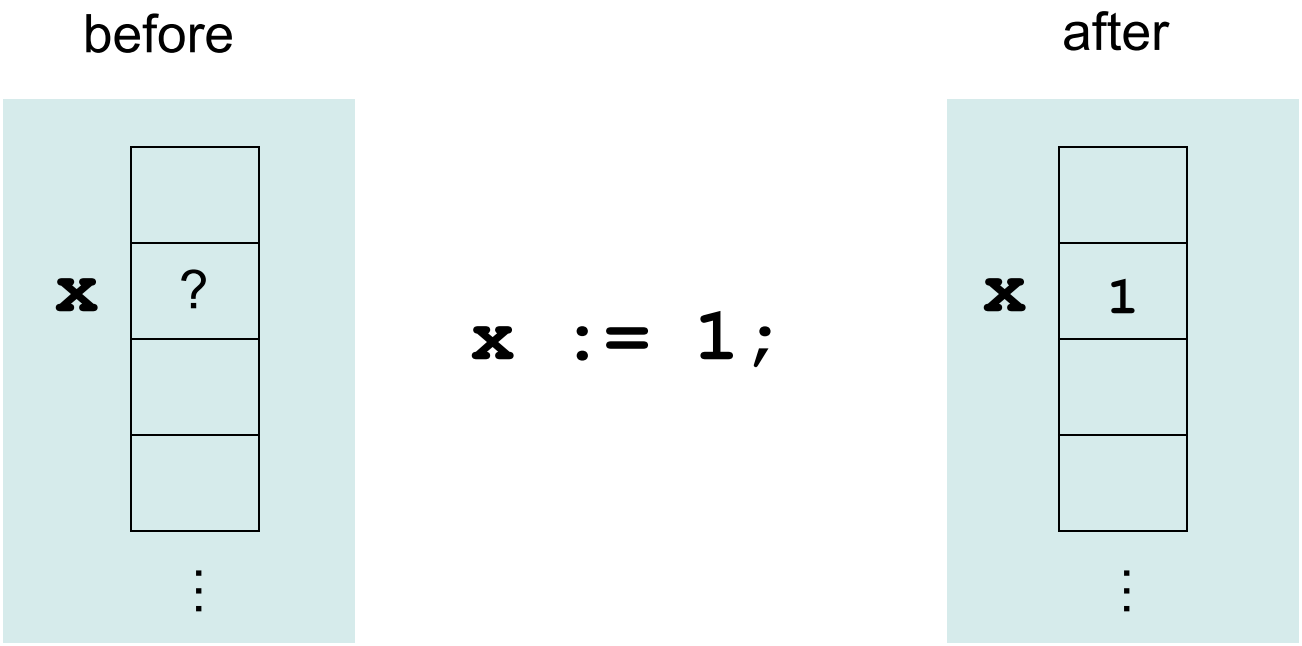
```
- x := 1;
val it = () : unit
```

why **`unit`**?

What is the meaning of "`x:=e`"?

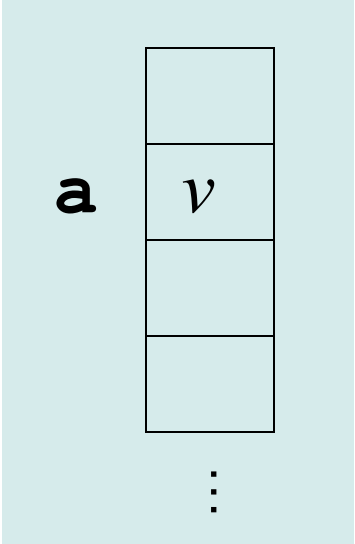


What is the meaning of "`x:=e`"?



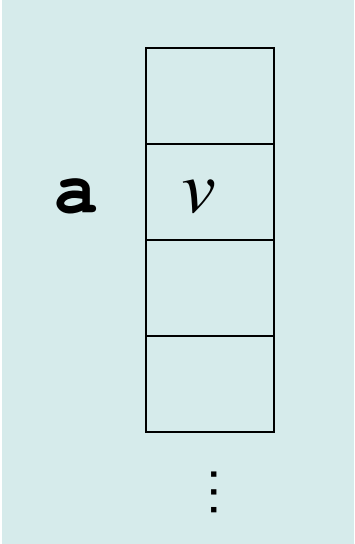
It takes a "Store" as input and returns a "Store" as output
I.e., it is a function of type "Store → Store"

How could one represent Store in Haskell?



…it is something that takes an address (**`a`**)
and returns a **`Value`** (v)

Store takes an address (**a**) and returns a **Value** (v)



…i.e, it is a function from Addresses to Values

Representing Store

- There are a number of choices for how we represent `Store`
- Think of addresses as variable names
  - i.e., `type Loc = String`
- Then Store can be implemented
  - As functions of type `Loc → Int`
    - `type Store = Loc → Int`
    - …or as association lists of type
    - `type Store = [(Loc,Int)]`

Store in Haskell (first pass)

```
type Loc = String

data Store = Mem [(Loc,Int)]

initStore = Mem []
```

A really simple language

```
data Imp = Assign Loc Int
         | Seq Imp Imp

c1 = Assign "x" 1
c2 = Assign "x" 2
c3 = Seq c1 c2
```

These are respectively:

```
x := 1
x := 2
x := 1 ; x := 2
```

The Imp interpreter

```
exec (Assign l i) (Mem s)
                = Mem ((l,i) : (dropfst l s))
exec (Seq c1 c2) mem0
                = let
                      mem1 = exec c1 mem0
                  in
                      exec c2 mem1
```

The Imp interpreter

Assignment just adds the "memory cell" to the Store. Think of **`dropfst`** as a garbage collector.

```
exec (Assign l i) (Mem s)
                = Mem ((l,i) : (dropfst l s))
exec (Seq c1 c2) mem0
                = let
                      mem1 = exec c1 mem0
                  in
                      exec c2 mem1
```

```
dropfst x [] = []
dropfst x ((y,v):rs) = if x==y
                       then dropfst x rs
                       else (y,v) : dropfst x rs
```

The Imp interpreter

"`;`" threads the Store through **`c1`** first then **`c2`**

```
exec (Assign l i) (Mem s)
                = Mem ((l,i) : (dropfst l s))
exec (Seq c1 c2) mem0
                = let
                      mem1 = exec c1 mem0
                  in
                      exec c2 mem1
```

1st **`c1`**,
2nd **`c2`**

Extension: multiple return values

- We consider two extensions to this language
  - return values – i.e., what if you want to define “+”?
  - errors – i.e., what happens when something goes wrong?
    - e.g., when something isn’t declared.

returning multiple values

- Currently, **`exec : Imp -> Store -> Store`**
- How would we add arithmetic?
  - **`exec (Litint i) mem`**$_{\mathbf{i}}$ = ?
  - **`exec (Var x) mem`**$_{\mathbf{i}}$ = ?
  - **`exec (Add i1 i2) mem`**$_{\mathbf{i}}$ = ?
  - Want the "?" to be an int, but doesn't type check.
- Idea: change **`exec`** to return two values
  - **`exec : Imp -> Store -> (Value , Store)`**

Extending the Abstract Syntax and adding Values

Add some new abstract syntax

```
data Imp = Assign Loc Int | Seq Imp Imp
         | Litint Int | Add Imp Imp | Var String
```

…and values

```
data Value = NilVal | I Int
```

As in ML, expressions can have side effects

```
- val x = ref 1;
val x = ref 1 : int ref
- (x := 3; !x) + 5;
val it = 8 : int
```

How do we represent this as an **Imp**?

```
Add (Seq (Assign "x" 3)) (Var "x"),Litint 5)
```

Two cases

```
data Value = NilVal | I Int

exec :: Imp -> Store -> (Value, Store)
exec (Assign l i) (Mem s)
                    = (NilVal, Mem ((l,i) : (dropfst l s)))
  …
exec (Litint i) mem = (I i, mem)
```

In these cases, the “action” occurs in different components of the returned pair.

All cases

```
exec (Assign l i) (Mem s)
                = (NilVal, Mem ((l,i) : (dropfst l s)))
exec (Litint i) mem       = (I i, mem)

exec (Seq c1 c2) mem0  = let
                             (_,mem1) = exec c1 mem0
                           in
                             exec c2 mem1

exec (Add i1 i2) mem
                = let
                        (I v1, mem1) = exec i1 mem
                        (I v2, mem2) = exec i2 mem1
                  in
                       (I (v1 + v2), mem2)

exec (Var x) (Mem m) = (I i, Mem m)
             where Just i = lookup x m
```

Example

Let `xsto = Mem [("x",0)]`, then

```
Imp> exec (Var "x") xsto
       (I 0,Mem [("x",0)])
```

Why?

```
exec (Var "x") (Mem [("x",0)])
   = (I i, Mem [("x",0)])
       where Just i = lookup "x" [("x",0)])
   = (I 0, Mem [("x",0)])
```

Summary

We defined a simple language for imperative programs:

```
type Loc = String
data Imp = Assign Loc Int | Seq Imp Imp
c1 = Assign "x" 1
c2 = Assign "x" 2
c3 = Seq  c1 c2
```

Storage (aka “memory” or “state”) was defined as lists of memory cells:

```
data Store = Mem [(Loc,Int)]
initsto = Mem []
```

Summary

Then we defined **`exec`** as a function of type **`Imp->Store->Store`**

```
dropfst x [] = []
dropfst x ((y,v):rs) = if x==y
                       then dropfst x rs
                       else (y,v) : dropfst x rs

exec :: Imp -> Store -> Store
exec (Assign l i) (Mem s) = Mem ((l,i) : (dropfst l s))
exec (Seq c1 c2) mem0     = let
                                mem1 = exec c1 mem0
                            in
                                exec c2 mem1
```