State

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Definition

The rule of *referential transparency:*

$$\frac{e_1 \rightarrow^* v \quad e_2 \rightarrow^* v \quad f\,e_1 \rightarrow^* w}{f\,e_2 \rightarrow^* w}$$

- If you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program.
- e.g. `f(x) + f(x) == 2 * f(x)`
- You can prove this by induction, using the natural semantic rules from the previous lectures.

- You can use equational reasoning to make the following equivalence:

$$\mathtt{f}(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) \equiv \mathtt{if}\ e_1\ \mathtt{then}\ \mathtt{f}(e_2)\ \mathtt{else}\ \mathtt{f}(e_3)$$

```
x * (if foo then 20 / x else 23 / x) \emph{equivalent to}
if foo then 20 else 23   \emph{(well, mostly)}
```

- You have the basis now of many compiler optimization opportunities!

A Complication

```
# let counter = -- something
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
# counter ();;
- : int = 3
#
```

- Can we still use equational reasoning to talk about programs now?

A Counterexample

- `f(x) + f(x) == 2 * f(x)`

```
# 2 * counter ();;
- : int = 8
# counter () + counter ();;
- : int = 11
```

- Congratulations. You just broke mathematics.

Reference Operator

Transition Semantics

`ref` $v \to \$i$, where $\$i$ is a free location in the state, initialized to v.

$!\,\$i \;\to\; v$, if state location $\$i$ contains v

$\$i := v \;\to\; ()$, and state location $\$i$ is assigned v.

$();e \;\to\; e$

Note that references are different than pointers: once created, they cannot be moved, only assigned to and read from.

Natural Semantics

$\dfrac{e \Downarrow v}{\texttt{ref}\ e \Downarrow \$i}$, where $\$i$ is a free location in the state, initialized to v.

$\dfrac{e \Downarrow \$i}{!e \Downarrow v}$, if state location $\$i$ contains v.

$\dfrac{e_1 \Downarrow \$i \quad e_2 \Downarrow v}{e_1 := e_2 \Downarrow ()}$, and location $\$i$ is set to v.

$\dfrac{e_1 \Downarrow () \quad e_2 \Downarrow v}{e_1; e_2 \Downarrow v}$

Counter, Method 1

```
# let ct = ref 0;;
val ct : int ref = {contents=0}
# let counter () =
    ct := !ct + 1;
    !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Bad Things for Counter

`ct` is globally defined. Two bad things could occur because of this.

1. What if you already had a global variable `ct` defined?
   - Correct solution: use modules.
2. The Stupid User^{TM} might decide to change `ct` just for fun.
   - Now your counter won't work like it's supposed to...
   - Now you can't change the representation without getting tech support calls.
   - Remember the idea of *abstraction*.

Conclusions about State

State is bad because:

- it breaks our ability to use equational reasoning
- users can get to our global variables and change them without permission

State is good because:

- Certain constructs are almost impossible without state (e.g., Graphs)
- Our world is a stateful one

Local Variable Example

```
# let foo x =
    let a = 10 + 20 in
      a + x;;
val foo : int -> int = <fun>
# foo 15;;
- : int = 45
# foo 30;;
- : int = 60
```

How many times does the `10 + 20` get computed?

Global Variable Example

```
# let a = 10 + 20;;
val a : int = 30
# let foo x =
    a + x;;
val foo : int -> int = <fun>
# foo 15;;
- : int = 45
# foo 30;;
- : int = 60
```

How many times does the `10 + 20` get computed?

Encapsulated Variable Example

```
# let foo =
     let a = 10 + 20 in
        fun x -> a + x;;
val foo : int -> int = <fun>
# foo 15;;
- : int = 45
# foo 30;;
- : int = 60
```

How many times does the `10 + 20` get computed?

Using local state

```
# let counter =
    let ct = ref 0 in
    fun () -> ct := !ct + 1; !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

- This protects `ct`, making it available only to `counter`.

Bad Pun

```
 # fun twice f x = f (f x)
 # twice counter () + twice counter ();;
 res4 : Int = 6
 # twice counter () + twice counter ();;
 res4 : Int = 14
```

- Function `twice` is the Church numeral for 2.
- You know what this means, right?

Bad Pun

```
 # fun twice f x = f (f x)
 # twice counter () + twice counter ();;
 res4 : Int = 6
 # twice counter () + twice counter ();;
 res4 : Int = 14
```

- Function `twice` is the Church numeral for 2.
- You know what this means, right?
- It means that you should never mix Church and state!

Random Number Generators

```
# let mkRandom s =
    let s = ref s in
      fun () -> s := (!s * 541 + 5) mod 1024; !s;;
 val mkRandom : int ref -> unit -> int = <fun>
# let rnd0 = mkRandom (ref 1);;
val rnd0 : unit -> int = <fun>
# rnd0 ();;
- : int = 546
# rnd0 ();;
- : int = 479
# rnd0 ();;
- : int = 72
```


Function Tuples

```
# let (counter, reset) =
    let ct = ref 0 in
      (fun () -> ct := !ct + 1; !ct),
      (fun nv -> ct := nv);;
val counter : unit -> int = <fun>
val reset : int -> unit = <fun>
# counter ();;
- : int = 1
# reset 5;;
- : unit = ()
# counter ();;
- : int = 6
```


Passing Counters Around

```
# let enumerate lst (ctfun, rsfun) =
    rsfun 0;
    List.map (fun x -> (ctfun (), x)) lst;;
val enumerate : 'a list ->
  (unit -> 'b) * (int -> 'c) -> ('b * 'a) list = <fun>
# enumerate ["hello";"there";"class"]
            (counter, reset);;
- : (int * string) list = [1, "hello"; 2, "there";
                           3, "class"]
#
```

- We can give the counter to another function.
- What could be problematic about this?