State

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
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. \(f(x) + f(x) = 2 \times f(x)\)
- 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:

\[ f(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \equiv \text{if } e_1 \text{ then } f(e_2) \text{ else } f(e_3) \]

\[ 1 \text{x } \ast (\text{if } \text{foo } \text{then } 20 \text{ / x } \text{else } 23 \text{ / x}) \quad \text{-- equivalent to} \]
\[ 2 \text{if } \text{foo } \text{then } 20 \text{ else } 23 \quad \text{-- well, mostly} \]

You have the basis now of many compiler optimization opportunities!
A Complication

```ocaml
# 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

- $f(x) + f(x) = 2 \cdot f(x)$

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

- Congratulations. You just broke mathematics.
**Reference Operator**

**Transition Semantics**

\[ \text{ref } v \rightarrow i, \text{ where } i \text{ is a free location in the state, initialized to } v. \]

\[ ! i \rightarrow v, \text{ if state location } i \text{ contains } v \]

\[ i := v \rightarrow (), \text{ and state location } i \text{ is assigned } v. \]

\[ (); e \rightarrow e \]

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

\[
\text{ref } e \downarrow i
\]

\[
e \downarrow v, \text{ where } i \text{ is a free location in the state, initialized to } v.
\]

\[
e \downarrow i, \text{ if state location } i \text{ contains } v.
\]

\[
!e \downarrow v
\]

\[
e_1 \downarrow i \quad e_2 \downarrow v, \text{ and location } i \text{ is set to } v.
\]

\[
e_1 := e_2 \downarrow ()
\]

\[
e_1 \downarrow () \quad e_2 \downarrow v
\]

\[
e_1; e_2 \downarrow v
\]
Counter, Method 1

```ocaml
# 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

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

1. What if you already had a global variable c_t defined?
   - Correct solution: use modules.

2. The Stupid User\textsuperscript{TM} might decide to change c_t 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

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

1. \texttt{# fun twice f x = f (f x)}
2. \texttt{# twice counter () + twice counter ();}
3. \texttt{res4 : Int = 6}
4. \texttt{# twice counter () + twice counter ();}
5. \texttt{res4 : Int = 14}

- Function \texttt{twice} is the Church numeral for 2.
- You know what this means, right?
Random Number Generators

```ocaml
# 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

```ocaml
# 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
```