State

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Describe the property of *referential transparency*.
- Explain how stateful computations complicate the meaning of programs.
- Use OCaml's references to model state.

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) -- equivalent to
if foo then 20 else 23  -- 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

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

$!\ \$i \ \rightarrow\ v$, if state location $\$i$ contains v.

$\$i := v \ \rightarrow\ ()$, and state location $\$i$ 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

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

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

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

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

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.