

CIS 500  
Software Foundations  
Fall 2006

September 27

# Programming in the Lambda-Calculus, Continued

## Testing booleans

---

Recall:

```
tru  =  $\lambda t. \lambda f. t$   
fls  =  $\lambda t. \lambda f. f$ 
```

We showed last time that, if  $b$  is a boolean (i.e., it behaves like either `tru` or `fls`), then, for any values  $v$  and  $w$ , either

$$b \ v \ w \longrightarrow^* \ v$$

(if  $b$  behaves like `tru`) or

$$b \ v \ w \longrightarrow^* \ w$$

(if  $b$  behaves like `fls`).

## Testing booleans

---

But what if we apply a boolean to terms that are *not* values?

E.g., what is the result of evaluating

```
tru c0 omega ?
```

## Testing booleans

---

But what if we apply a boolean to terms that are *not* values?

E.g., what is the result of evaluating

```
tru c0 omega ?
```

Not what we want!

## A better way

---

A dummy “unit value,” for forcing evaluation of thunks:

```
unit = λx. x
```

A “conditional function”:

```
test = λb. λt. λf. b t f unit
```

If  $b$  is a boolean (i.e., it behaves like either `tru` or `fls`), then, for arbitrary *terms*  $s$  and  $t$ , either

$$b (\lambda\text{dummy}. s) (\lambda\text{dummy}. t) \longrightarrow^* s$$

(if  $b$  behaves like `tru`) or

$$b (\lambda\text{dummy}. s) (\lambda\text{dummy}. t) \longrightarrow^* t$$

(if  $b$  behaves like `fls`).

## Review: The Z Operator

---

In the last lecture, we defined an operator `Z` that calculates the “fixed point” of a function it is applied to:

$$\begin{aligned} & \text{z} \\ & = \\ & \lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y \end{aligned}$$

That is, `z f v`  $\longrightarrow^*$  `f (z f) v`.

(N.b.: I’m writing it with a lower-case `z` today so that code snippets in the lecture notes can literally be typed into the `fulluntyped` interpreter, which expects identifiers to begin with lowercase letters.)

## Factorial

---

As an example, we defined the factorial function in lambda-calculus as follows:

```
fact    =    z ( λfct.  
                λn.  
                  if n=0 then 1  
                  else n * (fct (pred n)) )
```

For the sake of the example, we used “regular” booleans, numbers, etc.

I claimed that all this could be translated “straightforwardly” into the pure lambda-calculus.

Let’s do this.



# Factorial

---

```
badfact =  
  z (λfct.  
    λn.  
      iszro n  
        c1  
        (times n (fct (prd n))))
```

Why is this not what we want?

# Factorial

---

```
badfact =  
  z (λfct.  
    λn.  
      iszro n  
        c1  
        (times n (fct (prd n))))
```

Why is this not what we want?

(Hint: What happens when we evaluate `badfact c0`?)

# Factorial

---

A better version:

```
fact =  
  fix (λfct.  
      λn.  
        test (iszro n)  
              (λdummy. c1)  
              (λdummy. (times n (fct (prd n))))))
```

# Displaying numbers

---

fact c6 →\*

## Displaying numbers

---

fact c6  $\rightarrow^*$

```
(λs. λz.
  s ((λs. λz.
    s ((λs. λz.
      s ((λs. λz.
        s ((λs. λz.
          s ((λs. λz.z)
            s z))
          s z))
        s z))
      s z))
    s z))
  s z))
```

Ugh!

## Displaying numbers

---

If we enrich the pure lambda-calculus with “regular numbers,” we can display church numerals by converting them to regular numbers:

```
realnat = λn. n (λm. succ m) 0
```

Now:

```
realnat (times c2 c2)
      ↓*
succ (succ (succ (succ zero))).
```

## Displaying numbers

---

Alternatively, we can convert a few specific numbers to the form we want like this:

```
whack =
  λn. (equal n c0) c0
      ((equal n c1) c1
       ((equal n c2) c2
        ((equal n c3) c3
         ((equal n c4) c4
          ((equal n c5) c5
           ((equal n c6) c6
            n))))))
```

Now:

```
whack (fact c3)
      →*
λs. λz. s (s (s (s (s z))))))
```

# A Larger Example



In the second homework assignment, we saw how to encode an infinite stream as a thunk yielding a pair of a head element and another thunk representing the rest of the stream. The same encoding also works in the lambda-calculus.

Head and tail functions for streams:

```
streamhd = λs. fst (s unit)
streamtl = λs. snd (s unit)
```

A stream of increasing numbers:

```
upfrom =  
  fix  
    (λr.  
      λn.  
        λdummy.  
          pair n (r (scc n)))
```

Some tests:

```
whack (streamhd (upfrom c0))  
      →* c0
```

```
whack (streamhd (streamtl (upfrom c0)))  
      →* c2
```

```
whack (streamhd (streamtl (streamtl (upfrom c0))))  
      →* c4
```

## Mapping over streams:

```
streammap =  
  fix  
    (λsm.  
      λf.  
        λs.  
          λdummy.  
            pair (f (streamhd s)) (sm f (streamtl s)))
```

## Some tests:

```
evens = streammap double (upfrom c0);  
whack (streamhd evens);  
  /* yields c0 */  
whack (streamhd (streamtl evens));  
  /* yields c2 */  
whack (streamhd (streamtl (streamtl evens)));  
  /* yields c4 */
```

# Equivalence of Lambda Terms

## Representing Numbers

---

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s (s z)$$

$$c_3 = \lambda s. \lambda z. s (s (s z))$$

Other lambda-terms represent common operations on numbers:

$$scc = \lambda n. \lambda s. \lambda z. s (n s z)$$

## Representing Numbers

---

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s (s z)$$

$$c_3 = \lambda s. \lambda z. s (s (s z))$$

Other lambda-terms represent common operations on numbers:

$$scc = \lambda n. \lambda s. \lambda z. s (n s z)$$

In what sense can we say this representation is “correct”?  
In particular, on what basis can we argue that `scc` on church numerals corresponds to ordinary successor on numbers?

## The naive approach

---

One possibility:

For each  $n$ , the term  $scc\ c_n$  evaluates to  $c_{n+1}$ .

## The naive approach... doesn't work

---

One possibility:

For each  $n$ , the term `scc cn` evaluates to `cn+1`.

Unfortunately, this is false.

E.g.:

$$\begin{aligned} \text{scc } c_2 &= (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s (s z)) \\ &\longrightarrow \lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z) \\ &\neq \lambda s. \lambda z. s (s (s z)) \\ &= c_3 \end{aligned}$$



## A better approach

---

Recall the intuition behind the church numeral representation:

- ▶ a number  $n$  is represented as a term that “does something  $n$  times to something else”
- ▶ `scc` takes a term that “does something  $n$  times to something else” and returns a term that “does something  $n + 1$  times to something else”

I.e., what we really care about is that `scc c2` behaves the same as `c3` when applied to two arguments.

$$\begin{aligned}
\text{scc } c_2 \ v \ w &= (\lambda n. \lambda s. \lambda z. s (n \ s \ z)) (\lambda s. \lambda z. s (s \ z)) \ v \ w \\
&\longrightarrow (\lambda s. \lambda z. s ((\lambda s. \lambda z. s (s \ z)) \ s \ z)) \ v \ w \\
&\longrightarrow (\lambda z. v ((\lambda s. \lambda z. s (s \ z)) \ v \ z)) \ w \\
&\longrightarrow v ((\lambda s. \lambda z. s (s \ z)) \ v \ w) \\
&\longrightarrow v ((\lambda z. v (v \ z)) \ w) \\
&\longrightarrow v (v (v \ w))
\end{aligned}$$

$$\begin{aligned}
c_3 \ v \ w &= (\lambda s. \lambda z. s (s (s \ z))) \ v \ w \\
&\longrightarrow (\lambda z. v (v (v \ z))) \ w \\
&\longrightarrow v (v (v \ w))
\end{aligned}$$

## A general question

---

We have argued that, although  $scc$   $c_2$  and  $c_3$  do not evaluate to the same thing, they are nevertheless “behaviorally equivalent.”

What, precisely, does behavioral equivalence mean?

## Intuition

---

Roughly,

“terms  $s$  and  $t$  are behaviorally equivalent”

should mean:

“there is no ‘test’ that distinguishes  $s$  and  $t$  — i.e., no way to put them in the same context and observe different results.”

## Intuition

---

Roughly,

“terms  $s$  and  $t$  are behaviorally equivalent”

should mean:

“there is no ‘test’ that distinguishes  $s$  and  $t$  — i.e., no way to put them in the same context and observe different results.”

To make this precise, we need to be clear what we mean by a *testing context* and how we are going to *observe* the results of a test.

## Examples

---

```
tru = λt. λf. t
tru' = λt. λf. (λx.x) t
fls = λt. λf. f
omega = (λx. x x) (λx. x x)
poisonpill = λx. omega
placebo = λx. tru
Yf = (λx. f (x x)) (λx. f (x x))
```

Which of these are behaviorally equivalent?

## Observational equivalence

---

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms  $s$  and  $t$  are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

I.e., we “observe” a term’s behavior simply by running it and seeing if it halts.

## Observational equivalence

---

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms  $s$  and  $t$  are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

I.e., we “observe” a term’s behavior simply by running it and seeing if it halts.

Aside:

- ▶ Is observational equivalence a decidable property?



## Observational equivalence

---

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms  $s$  and  $t$  are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

I.e., we “observe” a term’s behavior simply by running it and seeing if it halts.

Aside:

- ▶ Is observational equivalence a decidable property?
- ▶ Does this mean the definition is ill-formed?

## Examples

---

- ▶ `omega` and `tru` are *not* observationally equivalent

## Examples

---

- ▶ `omega` and `tru` are *not* observationally equivalent
- ▶ `tru` and `fls` are observationally equivalent

## Behavioral Equivalence

---

This primitive notion of observation now gives us a way of “testing” terms for behavioral equivalence

Terms  $s$  and  $t$  are said to be *behaviorally equivalent* if, for every finite sequence of values  $v_1, v_2, \dots, v_n$ , the applications

$$s \ v_1 \ v_2 \ \dots \ v_n$$

and

$$t \ v_1 \ v_2 \ \dots \ v_n$$

are observationally equivalent.

## Examples

---

These terms are behaviorally equivalent:

```
tru = λt. λf. t
tru' = λt. λf. (λx.x) t
```

So are these:

```
omega = (λx. x x) (λx. x x)
Yf = (λx. f (x x)) (λx. f (x x))
```

These are not behaviorally equivalent (to each other, or to any of the terms above):

```
fls = λt. λf. f
poisonpill = λx. omega
placebo = λx. tru
```

## Proving behavioral equivalence

---

Given terms  $s$  and  $t$ , how do we *prove* that they are (or are not) behaviorally equivalent?

## Proving behavioral inequivalence

---

To prove that  $s$  and  $t$  are *not* behaviorally equivalent, it suffices to find a sequence of values  $v_1 \dots v_n$  such that one of

$$s \ v_1 \ v_2 \ \dots \ v_n$$

and

$$t \ v_1 \ v_2 \ \dots \ v_n$$

diverges, while the other reaches a normal form.

## Proving behavioral inequivalence

---

Example:

- ▶ the single argument `unit` demonstrates that `fls` is not behaviorally equivalent to `poisonpill`:

```
fls unit
= (λt. λf. f) unit
  →* λf. f
```

```
poisonpill unit
diverges
```



## Proving behavioral inequivalence

---

Example:

- ▶ the argument sequence `(λx. x) poisonpill (λx. x)` demonstrate that `tru` is not behaviorally equivalent to `fls`:

$$\begin{aligned} & \text{tru } (\lambda x. x) \text{ poisonpill } (\lambda x. x) \\ & \quad \longrightarrow^* (\lambda x. x)(\lambda x. x) \\ & \quad \quad \longrightarrow^* \lambda x. x \end{aligned}$$
$$\begin{aligned} & \text{fls } (\lambda x. x) \text{ poisonpill } (\lambda x. x) \\ & \longrightarrow^* \text{poisonpill } (\lambda x. x), \text{ which diverges} \end{aligned}$$

## Proving behavioral equivalence

---

To prove that  $s$  and  $t$  are behaviorally equivalent, we have to work harder: we must show that, for every sequence of values  $v_1 \dots v_n$ , either both

$s \ v_1 \ v_2 \ \dots \ v_n$

and

$t \ v_1 \ v_2 \ \dots \ v_n$

diverge, or else both reach a normal form.

How can we do this?

## Proving behavioral equivalence

---

In general, such proofs require some additional machinery that we will not have time to get into in this course (so-called *applicative bisimulation*). But, in some cases, we can find simple proofs.

*Theorem:* These terms are behaviorally equivalent:

$$\begin{aligned}\text{tru} &= \lambda t. \lambda f. t \\ \text{tru}' &= \lambda t. \lambda f. (\lambda x.x) t\end{aligned}$$

*Proof:* Consider an arbitrary sequence of values  $v_1 \dots v_n$ .

- ▶ For the case where the sequence has just one element (i.e.,  $n = 1$ ), note that both  $\text{tru } v_1$  and  $\text{tru}' v_1$  reach normal forms after one reduction step.
- ▶ For the case where the sequence has more than one element (i.e.,  $n > 1$ ), note that both  $\text{tru } v_1 v_2 v_3 \dots v_n$  and  $\text{tru}' v_1 v_2 v_3 \dots v_n$  reduce (in two steps) to  $v_1 v_3 \dots v_n$ . So either both normalize or both diverge.

## Proving behavioral equivalence

---

*Theorem:* These terms are behaviorally equivalent:

$$\begin{aligned}\omega &= (\lambda x. x x) (\lambda x. x x) \\ Y_f &= (\lambda x. f (x x)) (\lambda x. f (x x))\end{aligned}$$

*Proof:* Both

$$\omega v_1 \dots v_n$$

and

$$Y_f v_1 \dots v_n$$

diverge, for every sequence of arguments  $v_1 \dots v_n$ .

# Inductive Proofs about the Lambda Calculus

## Two induction principles

---

Like before, we have two ways to prove that properties are true of the untyped lambda calculus.

- ▶ Structural induction on terms
- ▶ Induction on a derivation of  $t \longrightarrow t'$ .

Let's look at an example of each.

## Structural induction on terms

---

To show that a property  $\mathcal{P}$  holds for all lambda-terms  $t$ , it suffices to show that

- ▶  $\mathcal{P}$  holds when  $t$  is a variable;
- ▶  $\mathcal{P}$  holds when  $t$  is a lambda-abstraction  $\lambda x. t_1$ , assuming that  $\mathcal{P}$  holds for the immediate subterm  $t_1$ ; and
- ▶  $\mathcal{P}$  holds when  $t$  is an application  $t_1 t_2$ , assuming that  $\mathcal{P}$  holds for the immediate subterms  $t_1$  and  $t_2$ .

## Structural induction on terms

---

To show that a property  $\mathcal{P}$  holds for all lambda-terms  $t$ , it suffices to show that

- ▶  $\mathcal{P}$  holds when  $t$  is a variable;
- ▶  $\mathcal{P}$  holds when  $t$  is a lambda-abstraction  $\lambda x. t_1$ , assuming that  $\mathcal{P}$  holds for the immediate subterm  $t_1$ ; and
- ▶  $\mathcal{P}$  holds when  $t$  is an application  $t_1 t_2$ , assuming that  $\mathcal{P}$  holds for the immediate subterms  $t_1$  and  $t_2$ .

N.b.: The variant of this principle where “immediate subterm” is replaced by “arbitrary subterm” is also valid. (Cf. *ordinary induction vs. complete induction* on the natural numbers.)



## An example of structural induction on terms

---

Define the set of *free variables* in a lambda-term as follows:

$$FV(x) = \{x\}$$

$$FV(\lambda x. t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

Define the *size* of a lambda-term as follows:

$$size(x) = 1$$

$$size(\lambda x. t_1) = size(t_1) + 1$$

$$size(t_1 t_2) = size(t_1) + size(t_2) + 1$$

*Theorem:*  $|FV(t)| \leq size(t)$ .

## An example of structural induction on terms

---

*Theorem:*  $|FV(t)| \leq size(t)$ .

*Proof:* By induction on the structure of  $t$ .

▶ If  $t$  is a variable, then  $|FV(t)| = 1 = size(t)$ .

▶ If  $t$  is an abstraction  $\lambda x. t_1$ , then

$$\begin{aligned} & |FV(t)| \\ = & |FV(t_1) \setminus \{x\}| && \text{by defn} \\ \leq & |FV(t_1)| && \text{by arithmetic} \\ \leq & size(t_1) && \text{by induction hypothesis} \\ \leq & size(t_1) + 1 && \text{by arithmetic} \\ = & size(t) && \text{by defn.} \end{aligned}$$

## An example of structural induction on terms

---

*Theorem:*  $|FV(t)| \leq size(t)$ .

*Proof:* By induction on the structure of  $t$ .

- ▶ If  $t$  is an application  $t_1 t_2$ , then

$$\begin{aligned} & |FV(t)| \\ = & |FV(t_1) \cup FV(t_2)| && \text{by defn} \\ \leq & \max(|FV(t_1)|, |FV(t_2)|) && \text{by arithmetic} \\ \leq & \max(|size(t_1)|, |size(t_2)|) && \text{by IH and arithmetic} \\ \leq & |size(t_1)| + |size(t_2)| && \text{by arithmetic} \\ \leq & |size(t_1)| + |size(t_2)| + 1 && \text{by arithmetic} \\ = & size(t) && \text{by defn.} \end{aligned}$$

## Induction on derivations

---

Recall that the reduction relation is defined as the smallest binary relation on terms satisfying the following rules:

$$(\lambda x. t_{12}) v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \quad (\text{E-APP2})$$

## Induction on derivations

---

Induction principle for the small-step evaluation relation.

To show that a property  $\mathcal{P}$  holds for all derivations of  $t \longrightarrow t'$ , it suffices to show that

- ▶  $\mathcal{P}$  holds for all derivations that use the rule E-AppAbs;
- ▶  $\mathcal{P}$  holds for all derivations that end with a use of E-App1 assuming that  $\mathcal{P}$  holds for all subderivations; and
- ▶  $\mathcal{P}$  holds for all derivations that end with a use of E-App2 assuming that  $\mathcal{P}$  holds for all subderivations.

## Example

---

Theorem: if  $t \longrightarrow t'$  then  $FV(t) \supseteq FV(t')$ .

## Induction on derivations

---

We must prove, for all derivations of  $t \longrightarrow t'$ , that  $FV(t) \supseteq FV(t')$ .

There are three cases.

## Induction on derivations

---

We must prove, for all derivations of  $t \longrightarrow t'$ , that  $FV(t) \supseteq FV(t')$ .

There are three cases.

- ▶ If the derivation of  $t \longrightarrow t'$  is just a use of E-AppAbs, then  $t$  is  $(\lambda x. t_1)v$  and  $t'$  is  $[x \mapsto v]t_1$ . Reason as follows:

$$\begin{aligned} FV(t) &= FV((\lambda x. t_1)v) \\ &= FV(t_1) \setminus \{x\} \cup FV(v) \\ &\supseteq FV([x \mapsto v]t_1) \\ &= FV(t') \end{aligned}$$



- ▶ If the derivation ends with a use of E-App1, then  $t$  has the form  $t_1 t_2$  and  $t'$  has the form  $t'_1 t_2$ , and we have a subderivation of  $t_1 \longrightarrow t'_1$

By the induction hypothesis,  $FV(t_1) \supseteq FV(t'_1)$ . Now calculate:

$$\begin{aligned} FV(t) &= FV(t_1 t_2) \\ &= FV(t_1) \cup FV(t_2) \\ &\supseteq FV(t'_1) \cup FV(t_2) \\ &= FV(t'_1 t_2) \\ &= FV(t') \end{aligned}$$

- ▶ If the derivation ends with a use of E-App1, then  $t$  has the form  $t_1 t_2$  and  $t'$  has the form  $t'_1 t_2$ , and we have a subderivation of  $t_1 \longrightarrow t'_1$

By the induction hypothesis,  $FV(t_1) \supseteq FV(t'_1)$ . Now calculate:

$$\begin{aligned} FV(t) &= FV(t_1 t_2) \\ &= FV(t_1) \cup FV(t_2) \\ &\supseteq FV(t'_1) \cup FV(t_2) \\ &= FV(t'_1 t_2) \\ &= FV(t') \end{aligned}$$

- ▶ If the derivation ends with a use of E-App2, the argument is similar to the previous case.