Testing booleans

Recall:
\[
\text{tru} = \lambda t. \lambda f. t \\
\text{fls} = \lambda t. \lambda f. f
\]

We showed last time that, if \( b \) is a boolean (i.e., it behaves like either \( \text{tru} \) or \( \text{fls} \)), then, for any values \( v \) and \( w \), either
\[
\text{b \ v \ w} \rightarrow^* v
\]
(if \( b \) behaves like \( \text{tru} \)) or
\[
\text{b \ v \ w} \rightarrow^* w
\]
(if \( b \) behaves like \( \text{fls} \)).

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

E.g., what is the result of evaluating
\[
\text{tru} \ c0 \ omega
\]
Not what we want!

A better way

A dummy “unit value,” for forcing evaluation of thunks:
\[
\text{unit} = \lambda x. x
\]

A “conditional function”:
\[
\text{test} = \lambda b. \lambda t. \lambda f. b \ t \ f \ \text{unit}
\]

If \( b \) is a boolean (i.e., it behaves like either \( \text{tru} \) or \( \text{fls} \)), then, for arbitrary terms \( s \) and \( t \), either
\[
\text{b \ (lambda. \ s) \ (lambda. \ t)} \rightarrow^* s
\]
(if \( b \) behaves like \( \text{tru} \)) or
\[
\text{b \ (lambda. \ s) \ (lambda. \ t)} \rightarrow^* t
\]
(if \( b \) behaves like \( \text{fls} \)).
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:

$$Z = \lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$$

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

(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:

$$\text{fact} = z (\lambda fct. \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \times (fct (\text{pred } n)))$$

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.

Displaying numbers

$$\text{fact c6} \rightarrow^* (\lambda s. \lambda z. s ((\lambda s. \lambda z. s ((\lambda s. \lambda z. s ((\lambda s. \lambda z. s ((\lambda s. \lambda z. s ((\lambda s. \lambda z. s ((\lambda s. \lambda z. s (\lambda n. \text{test (iszro n) (\lambda dummy. c1) (\lambda dummy. (times n (fct (prd n))))))))))))))))))))$$

Ugh!

Factorial

$$\text{badfact} = z (\lambda fct. \lambda n. \text{iszro n c1 (times n (fct (prd n))))}$$

Why is this not what we want?

(Hint: What happens when we evaluate $\text{badfact c0}$?)

Factorial

$$\text{fact} = \text{fix (\lambda fct. \lambda n. \text{test (iszro n) (\lambda dummy. c1) (\lambda dummy. (times n (fct (prd n))))}})$$

A better version:

$$\text{fact} = \text{fix (\lambda fct. \lambda n. \text{test (iszro n) (\lambda dummy. c1) (\lambda dummy. (times n (fct (prd n)))))}$$

Displaying numbers

$$\text{fact c6} \rightarrow^*$$
If we enrich the pure lambda-calculus with “regular numbers,” we can display church numerals by converting them to regular numbers:

$$\text{realnat} = \lambda n. n (\lambda m. \text{succ} m) 0$$

Now:

$$\text{realnat} (\text{times} \ c2 \ c2) \to \ast \ \text{succ} (\text{succ} (\text{succ} (\text{succ} \ 0))).$$

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

$$\text{whack} = \lambda n. (\text{equal} \ n \ c0) \ c0 \\
\quad \quad (\text{equal} \ n \ c1) \ c1 \\
\quad \quad (\text{equal} \ n \ c2) \ c2 \\
\quad \quad (\text{equal} \ n \ c3) \ c3 \\
\quad \quad (\text{equal} \ n \ c4) \ c4 \\
\quad \quad (\text{equal} \ n \ c5) \ c5 \\
\quad \quad (\text{equal} \ n \ c6) \ c6 \\
\quad \quad n)$$

Now:

$$\text{whack} (\text{fact} \ c3) \to \ast \ \lambda s. \lambda z. s (s (s (s (s (s z)))))$$

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:

$$\text{streamhd} = \lambda s. \text{fst} \ (s \ \text{unit})$$

$$\text{streamtl} = \lambda s. \text{snd} \ (s \ \text{unit})$$

A stream of increasing numbers:

$$\text{upfrom} = \text{fix} (\lambda r. \lambda n. \lambda dummy. \ \text{pair} \ n \ (r \ (\text{succ} \ n)))$$

Some tests:

$$\text{whack} (\text{streamhd} \ (\text{upfrom} \ c0)) \to \ast \ c0$$

$$\text{whack} (\text{streamhd} \ (\text{streamtl} \ (\text{upfrom} \ c0))) \to \ast \ c2$$

$$\text{whack} (\text{streamhd} \ (\text{streamtl} \ (\text{streamtl} \ (\text{upfrom} \ c0)))) \to \ast \ c4$$
Mapping over streams:

\[
\text{streammap} = \text{fix} (\lambda sm. \lambda f. \lambda s. \lambda dummy. \text{pair} (f (\text{streamhd} s)) (sm f (\text{streamtl} s)))
\]

Some tests:

\[
evens = \text{streammap double (upfrom c0)};
\]

whack (\text{streamhd} evens);
/* yields c0 */

whack (\text{streamhd} (\text{streamtl} evens));
/* yields c2 */

whack (\text{streamhd} (\text{streamtl} (\text{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:

\[
scc = \lambda n. \lambda s. \lambda z. s (n s z)
\]

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... doesn’t work

One possibility:

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

Unfortunately, this is false.

E.g.:

\[
scc \ c_2 = (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s (s z))
\]

\[\quad\rightarrow \lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z)
\]

\[\quad\neq \lambda s. \lambda z. s (s (s z))
\]

\[\quad= c_3
\]
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”
- \( \text{succ} \) 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 \( \text{succ} \ c_2 \) behaves the same as \( c_3 \) when applied to two arguments.

\[
\text{succ} \ c_2 \ v \ w = (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s (s z)) v \ w
\]

\[
\text{succ} \ c_2 \ v \ w \rightarrow (\lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z)) v \ w
\]

\[
\text{succ} \ c_2 \ v \ w \rightarrow v ((\lambda s. \lambda z. s (s z)) v z)
\]

\[
\text{succ} \ c_2 \ v \ w \rightarrow v ((\lambda z. v (v z)) w)
\]

\[
\text{succ} \ c_2 \ v \ w \rightarrow v (v (v w))
\]

A general question

We have argued that, although \( \text{succ} \ c_2 \) 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.”

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

\[
\text{tru} = \lambda t. \lambda f. t
\]

\[
\text{tru}' = \lambda t. \lambda f. (\lambda x.x) t
\]

\[
\text{fls} = \lambda t. \lambda f. f
\]

\[
\text{omega} = (\lambda x. x x) (\lambda x. x x)
\]

\[
\text{poisonpill} = \lambda x. \text{omega}
\]

\[
\text{placebo} = \lambda x. \text{tru}
\]

\[
\text{Y} f = (\lambda x. f (x x)) (\lambda x. f (x x))
\]

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.

Aside:

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

Examples

▶ \( \omega \) and \( \text{tru} \) are not observationally equivalent
▶ \( \text{tru} \) and \( \text{fls} \) 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 \( v_1, v_2, \ldots, v_n \), the applications

\[
s \; v_1 \; v_2 \; \ldots \; v_n
\]

and

\[
t \; v_1 \; v_2 \; \ldots \; v_n
\]

are observationally equivalent.
Examples

These terms are behaviorally equivalent:
\[ \text{tru} = \lambda t. \lambda f. t \]
\[ \text{tru'} = \lambda t. \lambda f. (\lambda x. x) t \]

So are these:
\[ \omega = (\lambda x. x x) (\lambda x. x x) \]
\[ Y_f = (\lambda x. f (x x)) (\lambda x. f (x x)) \]

These are not behaviorally equivalent (to each other, or to any of the terms above):
\[ \text{fls} = \lambda t. \lambda f. f \]
\[ \text{poisonpill} = \lambda x. \omega \]
\[ \text{placebo} = \lambda x. \text{tru} \]

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 \( v_1 \ldots v_n \) such that one of
\[ s \ v_1 \ v_2 \ldots \ v_n \]
and
\[ t \ v_1 \ v_2 \ldots \ v_n \]
diverges, while the other reaches a normal form.

Example:
\[ \text{the single argument } \text{unit demonstrates that fls is not behaviorally equivalent to poisonpill:} \]
\[ \text{fls unit} \]
\[ = (\lambda t. \lambda f. f) \text{ unit} \]
\[ \longrightarrow' \lambda f. f \]
\[ \text{poisonpill unit} \]
\[ \text{diverges} \]

Example:
\[ \text{the argument sequence } (\lambda x. x) \text{ poisonpill } (\lambda x. x) \]
demonstrate that \( \text{tru} \) is not behaviorally equivalent to \( \text{fls} \):
\[ \text{tru} (\lambda x. x) \text{ poisonpill } (\lambda x. x) \]
\[ \longrightarrow' (\lambda x. x)(\lambda x. x) \]
\[ \longrightarrow' \lambda x. x \]
\[ \text{fls } (\lambda x. x) \text{ poisonpill } (\lambda x. x) \]
\[ \longrightarrow' \text{poisonpill } (\lambda x. x), \text{ which diverges} \]

Proving behavioral inequivalence

To prove that \( s \) and \( t \) are behaviorally equivalent, we have to work harder: we must show that, for every sequence of values \( v_1 \ldots v_n \), either both
\[ s \ v_1 \ v_2 \ldots \ v_n \]
and
\[ t \ v_1 \ v_2 \ldots \ v_n \]
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:

\[
\text{tru} = \lambda t. \lambda f. t \\
\text{tru}' = \lambda t. \lambda f. (\lambda x. x) t
\]

**Proof:** Consider an arbitrary sequence of values \(v_1 \ldots v_n\).

- 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 \(\text{tru} \ v_1 \ v_2 \ v_3 \ldots \ v_n\) and \(\text{tru}' \ v_1 \ v_2 \ v_3 \ldots \ v_n\) reduce (in two steps) to \(v_1 \ v_3 \ldots \ v_n\). So either both normalize or both diverge.

Proving behavioral equivalence

**Theorem:** These terms are behaviorally equivalent:

\[
\omega = (\lambda x. x \ x) (\lambda x. x \ x) \\
Y_f = (\lambda x. f (x \ x)) (\lambda x. f (x \ x))
\]

**Proof:** Both \(\omega \ v_1 \ldots v_n\) and \(Y_f \ v_1 \ldots v_n\) diverge, for every sequence of arguments \(v_1 \ldots v_n\).

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 \(t \xrightarrow{} t'\).

Let’s look at an example of each.

Structural induction on terms

To show that a property \(P\) holds for all lambda-terms \(t\), it suffices to show that

- \(P\) holds when \(t\) is a variable;
- \(P\) holds when \(t\) is a lambda-abstraction \(\lambda x. \ t_1\), assuming that \(P\) holds for the immediate subterm \(t_1\); and
- \(P\) holds when \(t\) is an application \(t_1 \ t_2\), assuming that \(P\) holds for the immediate subterms \(t_1\) and \(t_2\).

Structural induction on terms

To show that a property \(P\) holds for all lambda-terms \(t\), it suffices to show that

- \(P\) holds when \(t\) is a variable;
- \(P\) holds when \(t\) is a lambda-abstraction \(\lambda x. \ t_1\), assuming that \(P\) holds for the immediate subterm \(t_1\); and
- \(P\) holds when \(t\) is an application \(t_1 \ t_2\), assuming that \(P\) 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:

\[ \text{FV}(x) = \{x\} \]
\[ \text{FV}(\lambda x . t_1) = \text{FV}(t_1) \setminus \{x\} \]
\[ \text{FV}(t_1 . t_2) = \text{FV}(t_1) \cup \text{FV}(t_2) \]

Define the size of a lambda-term as follows:

\[
\begin{align*}
\text{size}(x) &= 1 \\
\text{size}(\lambda x . t_1) &= \text{size}(t_1) + 1 \\
\text{size}(t_1 . t_2) &= \text{size}(t_1) + \text{size}(t_2) + 1
\end{align*}
\]

**Theorem:** \( |\text{FV}(t)| \leq \text{size}(t) \).

**Proof:** By induction on the structure of \( t \).
- If \( t \) is a variable, then \( |\text{FV}(t)| = \text{size}(t) \).
- If \( t \) is an abstraction \( \lambda x . t_1 \), then
  \[
  |\text{FV}(t)| = \text{size}(t_1) \setminus \{x\} \leq \text{size}(t_1) \leq \text{size}(t_1) + 1 \leq \text{size}(t) \text{ by defn.}
  \]

Induction on derivations

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

\[
\begin{align*}
(\lambda x . t_{12}) v_2 &\rightarrow [x \mapsto v_2] t_{12} \quad \text{(E-AppAbs)} \\
&\quad \quad \quad \quad \quad \quad \text{(E-App1)} \\
t_1 &\rightarrow t_1' \\
t_1 t_2 &\rightarrow t_1' t_2 \\
v_1 &\rightarrow v_1 \\
v_1 t_2 &\rightarrow v_1 t_2 \quad \text{(E-App2)}
\end{align*}
\]

**Example**

**Theorem:** if \( t \rightarrow t' \) then \( \text{FV}(t) \supseteq \text{FV}(t') \).
We must prove, for all derivations of $t \rightarrow t'$, that $FV(t) \supseteq FV(t')$.

There are three cases.

▶ If the derivation of $t \rightarrow t'$ 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:

$$FV(t) = FV((\lambda x.t_1)v)$$
$$= FV(t_1)/\{x\} \cup FV(v)$$
$$\supseteq FV([x\mapsto v]t_1)$$
$$= FV(t')$$

▶ If the derivation ends with a use of E-App1, then $t$ has the form $t_1t_2$ and $t'$ has the form $t'_1t_2$, and we have a subderivation of $t_1 \rightarrow t'_1$.

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

$$FV(t) = FV(t_1t_2)$$
$$= FV(t_1) \cup FV(t_2)$$
$$\supseteq FV(t'_1) \cup FV(t_2)$$
$$= FV(t'_1t_2)$$
$$= FV(t')$$

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