Lecture 6

• full-beta reduction
• call-by-name fixed point combinator; haskell simulation
• formalizing lists
• behavioral and observational equivalence
• inductive proofs about lambda-calculus

• No Class Next Week (Columbus Day)
  • Office Hour on Tuesday Oct 15 4:30-6pm
  • No Office on Monday Oct 21
• Midterm: Tuesday Oct 22
Full-Beta Reduction

- can reduce anywhere

\[
\begin{array}{c}
t_1 \rightarrow t'_1 \\
t_1 t_2 \rightarrow t'_1 t_2
\end{array} \quad \frac{t_2 \rightarrow t'_2}{t_1 t_2 \rightarrow t_1 t'_2} \quad \frac{(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12}}{t_1 \rightarrow t'_1} \quad \frac{(\lambda x. x) \ (\lambda x. x) \ (\lambda z. (\lambda x. x) z)}{\lambda x. t_1 \rightarrow \lambda x. t'_1}
\]

\[
\begin{array}{c}
t_1 \rightarrow t'_1 \\
t_1 t_2 \rightarrow t'_1 t_2
\end{array} \quad \frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad \frac{(\lambda x. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12}}{t_1 \rightarrow t'_1} \quad \frac{(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12}}{\lambda x. t_1 \rightarrow \lambda x. t'_1}
\]
Fixed Point Combinator Y

- **call-by-value:**
  \[
  Y = \lambda f. (\lambda x. f (\lambda v. ((x x) v))) (\lambda x. f (\lambda v. ((x x) v)))
  \]

- **call-by-name:**
  \[
  Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))
  \]

\[
Y g
= (\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))) g
  \quad \text{(by definition of } Y)\]

\[
= (\lambda x . g (x x)) (\lambda x . g (x x))
  \quad \text{(}\beta\text{-reduction of } \lambda f: \text{ applied to } g)\]

\[
= g ((\lambda x . g (x x)) (\lambda x . g (x x)))
  \quad \text{(}\beta\text{-reduction of } \lambda x: \text{ applied inside})\]

\[
= g (Y g)
  \quad \text{(by second equality)}
\]

\[
Y g = g(Y g) = g(g(Y g)) = g(g(g(Y g))) = g(g(g(g(...))))
\]

\[
\text{fix :: } (a \rightarrow a) \rightarrow a
\]
\[
\text{fix } f = f (\text{fix } f)
\]

\[
f fct x = \text{if } x == 0 \text{ then } 1 \text{ else } x * fct (x-1)
  \quad \text{-- or}
\]
\[
f = \lambda fct \rightarrow \lambda n \rightarrow \text{if } n == 0 \text{ then } 1 \text{ else } n * fct (n-1)
\]
\[
\text{fact = fix } f
\]
both solutions are required for midterm 1

5.2.8 SOLUTION: This is the solution I had in mind:

\[
\begin{align*}
nil &= \lambda c. \lambda n. \ n; \\
cons &= \lambda h. \lambda t. \lambda c. \lambda n. \ c \ h \ (t \ c \ n); \\
head &= \lambda l. \ 1 \ (\lambda h. \lambda t. h) \ f1s; \\
tail &= \lambda l. \\
        \quad \text{fst} \ (l \ (\lambda x. \lambda p. \text{pair} \ (\text{snd} \ p) \ (\text{cons} \ x \ (\text{snd} \ p))) \ \text{(pair} \ nil \ nil)); \\
isnil &= \lambda l. 1 \ (\lambda h. \lambda t. \text{f1s}) \ \text{tru};
\end{align*}
\]

Here is a rather different approach:

\[
\begin{align*}
nil &= \text{pair} \ \text{tru} \ \text{tru} \ \text{tru}; \\
cons &= \lambda h. \lambda t. \text{pair} \ \text{f1s} \ (\text{pair} \ h \ t); \\
head &= \lambda z. \text{fst} \ (\text{snd} \ z); \\
tail &= \lambda z. \text{snd} \ (\text{snd} \ z); \\
isnil &= \text{fst};
\end{align*}
\]
5.2.11 Solution:

\[ ff = \lambda f. \lambda l. \]
\[ \quad \text{test (isnil } l) \]
\[ \quad \quad (\lambda x. c_0) (\lambda x. (\text{plus (head } l) (f (\text{tail } l)))) c_0; \]
\[ \text{sumlist} = \text{fix } ff; \]

\[ l = \text{cons } c_2 (\text{cons } c_3 (\text{cons } c_4 \text{ nil})); \]
\[ \text{equal (sumlist } l) c_9; \]
\[ \text{\textbullet{} } (\lambda x. \lambda y. x) \]

A list-summing function can also, of course, be written without using fix:

\[ \text{sumlist}' = \lambda l. \text{l plus } c_0; \]
\[ \text{equal (sumlist } l) c_9; \]
\[ \text{\textbullet{} } (\lambda x. \lambda y. x) \]
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:

\[ \text{succ} = \lambda n. \lambda s. \lambda z. s \, (n \, s \, z) \]
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:

\[ \text{sc}c = \lambda n. \lambda s. \lambda z. s \; (n \; s \; z) \]

In what sense can we say this representation is “correct”?

In particular, on what basis can we argue that \( \text{sc}c \) on church numerals corresponds to ordinary successor on numbers?
The naive approach

One possibility:

For each $n$, the term $\text{scc } c_n$ evaluates to $c_{n+1}$. 
The naive approach... doesn't work

One possibility:

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

Unfortunately, this is false.

E.g.:

\[
\begin{align*}
\text{sc} \, c_2 & = (\lambda n. \lambda s. \lambda z. s \,(n \, s \, z)) \,(\lambda s. \lambda z. s \,(s \, z)) \\
\rightarrow & \quad \lambda s. \lambda z. s \,((\lambda s. \lambda z. s \,(s \, z)) \, s \, z) \\
\neq & \quad \lambda s. \lambda z. s \, (s \,(s \, z)) \\
= & \quad c_3
\end{align*}
\]
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”

♦ \( \text{sc} \text{c} \) 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 \( \text{sc} \text{c} \text{c}_2 \) behaves the same as \( \text{c}_3 \) when applied to two arguments.
\[\text{sc} \ c_2 \ v \ w \quad = \quad (\lambda n. \ \lambda s. \ \lambda z. \ s \ (n \ s \ z)) \ (\lambda s. \ \lambda z. \ s \ (s \ z)) \ v \ w \\
\quad \to \quad (\lambda s. \ \lambda z. \ s \ ((\lambda s. \ \lambda z. \ s \ (s \ z)) \ s \ z)) \ v \ w \\
\quad \to \quad (\lambda z. \ v \ ((\lambda s. \ \lambda z. \ s \ (s \ z)) \ v \ z)) \ w \\
\quad \to \quad v \ ((\lambda s. \ \lambda z. \ s \ (s \ z)) \ v \ w) \\
\quad \to \quad v \ ((\lambda z. \ v \ (v \ z)) \ w) \\
\quad \to \quad v \ (v \ (v \ w))\]

\[\text{c}_3 \ v \ w \quad = \quad (\lambda s. \ \lambda z. \ s \ (s \ (s \ z))) \ v \ w \\
\quad \to \quad (\lambda z. \ v \ (v \ (v \ z))) \ w \\
\quad \to \quad v \ (v \ (v \ w)))\]
A More General Question

We have argued that, although $\text{scc } c_2$ 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 use them in the same context and obtain different results.
Some test cases

\[
\begin{align*}
\text{tru} &= \lambda t. \lambda f. \ t \\
\text{tru}' &= \lambda t. \lambda f. \ (\lambda x. x) \ t \\
\text{fls} &= \lambda t. \lambda f. \ f \\
\text{omega} &= (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \\
\text{poisonpill} &= \lambda x. \ \text{omega} \\
\text{placebo} &= \lambda x. \ \text{tru} \\
Y_f &= (\lambda x. \ f \ (x \ x)) \ (\lambda x. \ f \ (x \ x))
\end{align*}
\]

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 way of testing terms.

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 are divergent.

I.e., our primitive notion of “observing” a term’s behavior is simply running it on our abstract machine.
Observational equivalence

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

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 are divergent.

I.e., our primitive notion of “observing” a term’s behavior is simply running it on our abstract machine.

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 way of testing terms.

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 are divergent.

I.e., our primitive notion of “observing” a term’s behavior is simply running it on our abstract machine.

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 $v_1, v_2, \ldots, v_n$, 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:

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

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

\[ \text{fls} = \lambda t. \lambda f. \ f \]
\[ \text{poisonpill} = \lambda x. \text{omega} \]
\[ \text{placebo} = \lambda x. \text{tru} \]
Proving behavioral inequivalence

Example:

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

  \[
  \text{fls unit} \\
  = (\lambda t. \lambda f. f) \text{ unit} \\
  \rightarrow^* \lambda f. f
  \]

  \[
  \text{poisonpill unit} \\
  \text{diverges}
  \]

Example:

- the argument sequence \((\lambda x. x) \text{poisonpill} (\lambda x. x)\) demonstrate that tru is not behaviorally equivalent to fls:

  \[
  \text{tru} (\lambda x. x) \text{poisonpill} (\lambda x. x) \\
  \rightarrow^* (\lambda x. x)(\lambda x. x) \\
  \rightarrow^* \lambda x. x
  \]

  \[
  \text{fls} (\lambda x. x) \text{poisonpill} (\lambda x. x) \\
  \rightarrow^* \text{poisonpill} (\lambda x. x), \text{which diverges}
  \]
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 $t \rightarrow t'$. 

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 $t_2$. 

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.)
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:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(\lambda x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1 t_2) &= FV(t_1) \cup FV(t_2)
\end{align*}
\]

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

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

*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

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

\[
|FV(t)| \\
= |FV(t_1) \cup FV(t_2)| \quad \text{by defn} \\
\leq \max(|FV(t_1)| + |FV(t_2)|) \quad \text{by arithmetic} \\
\leq \max(|\text{size}(t_1)|, |\text{size}(t_2)|) \quad \text{by IH and arithmetic} \\
\leq |\text{size}(t_1)| + |\text{size}(t_2)| \quad \text{by arithmetic} \\
\leq |\text{size}(t_1)| + |\text{size}(t_2)| + 1 \quad \text{by arithmetic} \\
= \text{size}(t) \quad \text{by defn.}
\]
Induction on derivations

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

\[(\lambda x. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \quad (E\text{-AppAbs})\]

\[
\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (E\text{-App1})
\]

\[
\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad (E\text{-App2})
\]
Induction on derivations

Induction principle for the small-step evaluation relation.

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

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

Theorem: if $t \rightarrow t'$ then $FV(t) \supseteq FV(t')$. 
Induction on derivations

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

There are three cases.
Induction on derivations

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

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 $[x|\rightarrow v]t_1$. Reason as follows:

\[
\begin{align*}
\text{FV}(t) &= \text{FV}((\lambda x.t_1)v) \\
&= \text{FV}(t_1)/\{x\} \cup \text{FV}(v) \\
&\supseteq \text{FV}([x|\rightarrow v]t_1) \\
&= \text{FV}(t')
\end{align*}
\]
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 $t_1 \rightarrow t'_1$.

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

\[
FV(t) = FV(t_1) \cup FV(t_2) \\
\supseteq FV(t'_1) \cup FV(t_2) \\
= FV(t'_1) \cup FV(t_2) \\
= FV(t) \\
= FV(t')
\]
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 $t_1 \rightarrow t'_1$

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

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

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