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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \quad \text{(E-AppAbs)}$$

```
    t1 -> t1'
---------------            (E-Abs)
 \x.t1 -> \x.t1'
```

(λx.x) ((λx.x) (λz. (λx.x) z))

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \quad \text{(E-AppAbs)}$$

Fixed Point Combinator Y

- call-by-value: Y = λf.(λx.f (λv.((x x) v))) (λx.f (λv.((x x) v)))
- call-by-name: Y = λf.(λx.f (x x)) (λx.f (x x))

Y g
= (λf . (λx . f (x x)) (λx . f (x x))) g (by definition of **Y**)
= (λx . g (x x)) (λx . g (x x)) (β-reduction of λf: applied to g)
= g ((λx . g (x x)) (λx . g (x x))) (β-reduction of λx: applied inside)
= g (**Y** g) (by second equality)

Y g = g(Y g) = g(g(Y g)) = g(g(g(Y g))) = g(g(g(g(...))))

```
fix :: (a -> a) -> a
fix f = f (fix f)

f fct x = if x == 0 then 1 else x * fct (x-1)   -- or
f = \fct -> \n -> if n == 0 then 1 else n * fct (n-1)

fact = fix f
```

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

Lists

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```

both solutions are required for midterm 1

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

```
nil = λc. λn. n;
cons = λh. λt. λc. λn. c h (t c n);
head = λl. l (λh.λt.h) fls;
tail = λl.
          fst (l (λx. λp. pair (snd p) (cons x (snd p)))
                 (pair nil nil));
isnil = λl. l (λh.λt.fls) tru;
```

Here is a rather different approach:

```
nil = pair tru tru;
cons = λh. λt. pair fls (pair h t);
head = λz. fst (snd z);
tail = λz. snd (snd z);
isnil = fst;
```

List Sum

5.2.11 SOLUTION:

```
ff = λf. λl.
        test (isnil l)
           (λx. c0) (λx. (plus (head l) (f (tail l)))) c0;
sumlist = fix ff;

l = cons c2 (cons c3 (cons c4 nil));
equal (sumlist l) c9;
```

▸ (λx. λy. x)

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

```
sumlist' = λl. l plus c0;
equal (sumlist l) c9;
```

▸ (λx. λy. 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 = λs. λz. z

c_1 = λs. λz. s z

c_2 = λs. λz. s (s z)

c_3 = λs. λz. s (s (s z))

Other lambda-terms represent common operations on numbers:

scc = λn. λs. λz. 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 $\mathtt{scc}\ c_n$ evaluates to c_{n+1}.

The naive approach... doesn't work

One possibility:

For each n, the term $\texttt{scc}\ \texttt{c}_n$ evaluates to $\texttt{c}_{n+1}$.

Unfortunately, this is false.

E.g.:

$$\begin{array}{rcl} \texttt{scc c}_2 & = & (\lambda\texttt{n. }\lambda\texttt{s. }\lambda\texttt{z. s (n s z)) (}\lambda\texttt{s. }\lambda\texttt{z. s (s z))} \\ & \longrightarrow & \lambda\texttt{s. }\lambda\texttt{z. s ((}\lambda\texttt{s. }\lambda\texttt{z. s (s z)) s z)} \\ & \neq & \lambda\texttt{s. }\lambda\texttt{z. s (s (s z))} \\ & = & \texttt{c}_3 \end{array}$$

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` c_2 behaves the same as c_3 when applied to two arguments.

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

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

A More 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 use them in the same context and obtain different results.

Some test cases

```
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
Y_f = (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 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, ..., 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:

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

So are these:

```
omega = (λx. x x) (λx. x x)
```
Y_f `= (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 inequivalence

Example:

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

$$\begin{aligned} &\texttt{fls unit} \\ = &\ (\lambda\texttt{t. }\lambda\texttt{f. f) unit} \\ \longrightarrow^* &\ \lambda\texttt{f. f} \end{aligned}$$

$$\texttt{poisonpill unit}$$

diverges

Example:

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

$$\begin{aligned} &\texttt{tru } (\lambda\texttt{x. x})\texttt{ poisonpill } (\lambda\texttt{x. x}) \\ \longrightarrow^* &\ (\lambda\texttt{x. x})(\lambda\texttt{x. x}) \\ \longrightarrow^* &\ \lambda\texttt{x. x} \end{aligned}$$

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

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:

```
tru = λt. λf. t
tru' = λt. λf. (λx.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 tru v_1 and 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 tru v_1 v_2 v_3 ... v_n and tru′ v_1 v_2 v_3 ... v_n reduce (in two steps) to v_1 v_3 ... v_n. So either both normalize or both diverge.

Proving behavioral equivalence

Theorem: These terms are behaviorally equivalent:

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

Proof: Both

$$\texttt{omega}\ v_1 \ldots v_n$$

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 $\mathtt{t} \longrightarrow \mathtt{t}'$.

Let's look at an example of each.

Structural induction on terms

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

- $\mathcal{P}$ holds when $\mathtt{t}$ is a variable;
- $\mathcal{P}$ holds when $\mathtt{t}$ is a lambda-abstraction $\lambda\mathtt{x}.\ \mathtt{t}_1$, assuming that $\mathcal{P}$ holds for the immediate subterm $\mathtt{t}_1$; and
- $\mathcal{P}$ holds when $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, assuming that $\mathcal{P}$ holds for the immediate subterms $\mathtt{t}_1$ and $\mathtt{t}_2$.

Structural induction on terms

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

- $\mathcal{P}$ holds when $\mathtt{t}$ is a variable;
- $\mathcal{P}$ holds when $\mathtt{t}$ is a lambda-abstraction $\lambda\mathtt{x}.\ \mathtt{t}_1$, assuming that $\mathcal{P}$ holds for the immediate subterm $\mathtt{t}_1$; and
- $\mathcal{P}$ holds when $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, assuming that $\mathcal{P}$ holds for the immediate subterms $\mathtt{t}_1$ and $\mathtt{t}_2$.

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:

$$\begin{aligned} FV(\mathtt{x}) &= \{\mathtt{x}\} \\ FV(\lambda\mathtt{x}.\mathtt{t}_1) &= FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\ FV(\mathtt{t}_1\ \mathtt{t}_2) &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \end{aligned}$$

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

$$\begin{aligned} size(\mathtt{x}) &= 1 \\ size(\lambda\mathtt{x}.\mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\ size(\mathtt{t}_1\ \mathtt{t}_2) &= size(\mathtt{t}_1) + size(\mathtt{t}_2) + 1 \end{aligned}$$

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

An example of structural induction on terms

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on the structure of $\mathtt{t}$.

- If $\mathtt{t}$ is a variable, then $|FV(\mathtt{t})| = 1 = size(\mathtt{t})$.
- If $\mathtt{t}$ is an abstraction $\lambda\mathtt{x}.\ \mathtt{t}_1$, then

$$
\begin{array}{lll}
 & |FV(\mathtt{t})| & \\
= & |FV(\mathtt{t}_1) \setminus \{\mathtt{x}\}| & \text{by defn} \\
\leq & |FV(\mathtt{t}_1)| & \text{by arithmetic} \\
\leq & size(\mathtt{t}_1) & \text{by induction hypothesis} \\
\leq & size(\mathtt{t}_1) + 1 & \text{by arithmetic} \\
= & size(\mathtt{t}) & \text{by defn.}
\end{array}
$$

An example of structural induction on terms

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on the structure of $\mathtt{t}$.

- If $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, then

	$\lvert FV(\mathtt{t})\rvert$	
$=$	$\lvert FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)\rvert$	by defn
~~$\leq$~~	~~max~~$(\lvert FV(\mathtt{t}_1)\rvert$ ~~,~~ + $\lvert FV(\mathtt{t}_2)\rvert)$	by arithmetic — should be +
~~$\leq$~~	~~$max(\lvert size(\mathtt{t}_1)\rvert, \lvert size(\mathtt{t}_2)\rvert)$~~	~~by IH and arithmetic~~
$\leq$	$\lvert size(\mathtt{t}_1)\rvert + \lvert size(\mathtt{t}_2)\rvert$	by arithmetic — by IH
$\leq$	$\lvert size(\mathtt{t}_1)\rvert + \lvert size(\mathtt{t}_2)\rvert + 1$	by arithmetic
$=$	$size(\mathtt{t})$	by defn.

Induction on derivations

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

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Induction on derivations

Induction principle for the small-step evaluation relation.

To show that a property $\mathcal{P}$ holds for all derivations of $\mathtt{t} \longrightarrow \mathtt{t}'$, 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 $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

Induction on derivations

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

There are three cases.

Induction on derivations

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

There are three cases.

- If the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ is just a use of E-AppAbs, then $\mathtt{t}$ is $(\lambda\mathtt{x}.\mathtt{t}_1)\mathtt{v}$ and $\mathtt{t}'$ is $[\mathtt{x}\mapsto\mathtt{v}]\mathtt{t}_1$. Reason as follows:

$$
\begin{aligned}
FV(\mathtt{t}) &= FV((\lambda\mathtt{x}.\mathtt{t}_1)\mathtt{v}) \\
&= FV(\mathtt{t}_1)/\{\mathtt{x}\} \cup FV(\mathtt{v}) \\
&\supseteq FV([\mathtt{x}\mapsto\mathtt{v}]\mathtt{t}_1) \\
&= FV(\mathtt{t}')
\end{aligned}
$$

- If the derivation ends with a use of E-App1, then $\mathtt{t}$ has the form $\mathtt{t}_1\ \mathtt{t}_2$ and $\mathtt{t}'$ has the form $\mathtt{t}_1'\ \mathtt{t}_2$, and we have a subderivation of $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$

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

$$\begin{aligned} FV(t) &= FV(\mathtt{t}_1\ \mathtt{t}_2) \\ &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \\ &\supseteq FV(\mathtt{t}_1') \cup FV(\mathtt{t}_2) \\ &= FV(\mathtt{t}_1'\ \mathtt{t}_2) \\ &= FV(t') \end{aligned}$$

- If the derivation ends with a use of E-App1, then $\mathtt{t}$ has the form $\mathtt{t}_1\ \mathtt{t}_2$ and $\mathtt{t}'$ has the form $\mathtt{t}'_1\ \mathtt{t}_2$, and we have a subderivation of $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$

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

$$\begin{aligned} FV(t) &= FV(\mathtt{t}_1\ \mathtt{t}_2) \\ &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \\ &\supseteq FV(\mathtt{t}'_1) \cup FV(\mathtt{t}_2) \\ &= FV(\mathtt{t}'_1\ \mathtt{t}_2) \\ &= FV(t') \end{aligned}$$

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