Foundations of Software
Fall 2023

Week 4

Programming in the Lambda-Calculus, Continued

Recall: Church Booleans

$$\begin{aligned}\texttt{tru} &= \lambda\texttt{t}.\ \lambda\texttt{f}.\ \texttt{t}\\ \texttt{fls} &= \lambda\texttt{t}.\ \lambda\texttt{f}.\ \texttt{f}\end{aligned}$$

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

$$\texttt{b v w} \longrightarrow^* \texttt{v}$$

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

$$\texttt{b v w} \longrightarrow^* \texttt{w}$$

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

Booleans with "bad" arguments

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

E.g., what is the result of evaluating

$$\texttt{tru}\ \texttt{c}_0\ \texttt{omega}\ ?$$

Booleans with "bad" arguments

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

E.g., what is the result of evaluating

`tru` c_0 `omega` ?

Not what we want!

A better way

Wrap the branches in an abstraction, and use a dummy "unit value," to force evaluation of thunks:

```
unit  =  λx. x
```

Use a "conditional function":

```
test  =  λb. λt. λf. b t f unit
```

If `tru′` is or behaves like `tru`, `fls′` is or behaves like `fls`, and `s` and `t` are arbitrary terms then

```
test tru′ (λdummy. s) (λdummy. t) ⟶* s
test fls′ (λdummy. s) (λdummy. t) ⟶* t
```


Recall: 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  =
  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
```

That is, if z_f = `z f` then z_f `v` $\longrightarrow^*$ `f` z_f `v`.

Recall: Factorial

As an example, we defined the factorial function as follows:

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

For simplicity, we used primitive values from the calculus of numbers and booleans presented in week 2, and even used shortcuts like `1` and `*`.

As mentioned, this can be translated "straightforwardly" into the pure lambda-calculus. Let's do that.

Lambda calculus version of Factorial (not!)

Here is the naive translation:

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

Why is this not what we want?

Lambda calculus version of Factorial (not!)

Here is the naive translation:

```
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 c_0?)

Lambda calculus version of Factorial

A better version:

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


Displaying numbers

fact $c_3 \longrightarrow^*$

Displaying numbers

```
fact c₃ ⟶* (λs. λz.
             s ((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:

```
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 (s z)))))
```


Equivalence of Lambda Terms

Recall: Church Numerals

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)

Recall: Church Numerals

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)

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 c_n evaluates to c_{n+1}.

Unfortunately, this is false.
E.g.:

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

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.

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

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

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

```
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 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

$$\mathtt{s}\ \mathtt{v}_1\ \mathtt{v}_2\ \dots\ \mathtt{v}_n$$

and

$$\mathtt{t}\ \mathtt{v}_1\ \mathtt{v}_2\ \dots\ \mathtt{v}_n$$

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:

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

poisonpill unit
diverges

Proving behavioral inequivalence

Example:

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

$$\begin{array}{l} \texttt{tru (}\lambda\texttt{x. x) poisonpill (}\lambda\texttt{x. x)} \\ \quad \longrightarrow^* \texttt{(}\lambda\texttt{x. x)(}\lambda\texttt{x. x)} \\ \qquad \longrightarrow^* \lambda\texttt{x. x} \end{array}$$

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

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 $\texttt{v}_1 \ldots \texttt{v}_n$, either both

$$\texttt{s v}_1\ \texttt{v}_2\ \ldots\ \texttt{v}_n$$

and

$$\texttt{t v}_1\ \texttt{v}_2\ \ldots\ \texttt{v}_n$$

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:

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

Proof: Consider an arbitrary sequence of values $\texttt{v}_1 \ldots \texttt{v}_n$.

- For the case where the sequence has up to one element (i.e., $n \leq 1$), note that both `tru` / $\texttt{tru v}_1$ and $\texttt{tru}'$ / $\texttt{tru}'\ \texttt{v}_1$ reach normal forms after zero / one reduction steps.
- For the case where the sequence has more than one element (i.e., $n > 1$), note that both $\texttt{tru v}_1\ \texttt{v}_2\ \texttt{v}_3\ \ldots\ \texttt{v}_n$ and $\texttt{tru}'\ \texttt{v}_1\ \texttt{v}_2\ \texttt{v}_3\ \ldots\ \texttt{v}_n$ reduce to $\texttt{v}_1\ \texttt{v}_3\ \ldots\ \texttt{v}_n$. So either both normalize or both diverge.

Proving behavioral equivalence

Theorem: These terms are behaviorally equivalent:

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

Proof: Both

$$\texttt{omega v}_1 \ldots \texttt{v}_n$$

and

$$\texttt{Y}_f\ \texttt{v}_1 \ldots \texttt{v}_n$$

diverge, for every sequence of arguments $\texttt{v}_1 \ldots \texttt{v}_n$.

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:

$$\begin{array}{l} FV(x) = \{x\} \\ FV(\lambda x.t_1) = FV(t_1) \setminus \{x\} \\ FV(t_1\ t_2) = FV(t_1) \cup FV(t_2) \end{array}$$

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

$$\begin{array}{l} size(x) = 1 \\ size(\lambda x.t_1) = size(t_1) + 1 \\ size(t_1\ t_2) = size(t_1) + size(t_2) + 1 \end{array}$$

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

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

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

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

Induction on derivations

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

$$(\lambda x.t_1)\ v_2 \longrightarrow [x \mapsto v_2]t_1 \qquad \text{(E-AppAbs)}$$

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

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

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.

An example of induction on derivations

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

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

An example of induction on derivations

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

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. There are three cases:

An example of induction on derivations

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

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. 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:

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

An example of induction on derivations

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

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. There are three cases:

- 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:

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

- E-App2 is treated similarly.

Safety of the untyped lambda calculus

Clicker question: In your opinion, what is most important theorem we/you/the book proved so far about the untyped lambda calculus?

A. Confluence, i.e., a term evaluates in many steps to at most one normal form.

B. If $\mathtt{t}$ is closed and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}'$ is closed.

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

D. $FV([\mathtt{x} \to \mathtt{t}_2]\mathtt{t}_1) \subseteq FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \cup FV(\mathtt{t}_2)$

URL: `ttpoll.eu`
Session ID: `cs452`