Theory of Types and Programming Languages Fall 2022

Week 4

Programming in the Lambda-Calculus: Continued

Church Encoding

Recall Church encoding of *natural numbers*:

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$
$$\ldots$$

$$\texttt{succ n} = \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

Is that the only possible one? Can you think of another one?

Church vs Scott Encoding

Recall Church encoding of *natural numbers*:

```
c₀ = λs. λz. z
c₁ = λs. λz. s z
c₂ = λs. λz. s (s z)
c₃ = λs. λz. s (s (s z))
   ...

succ n = λs. λz. s (n s z)
```

Is that the only possible one? Can you think of another one?

Another encoding of data types, called *Scott* encoding:

```
   c₀' = λs. λz. z
succ' n = λs. λz. s n
```

Church vs Scott Encoding

Recall Church encoding of *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))
  ...

succ n = λs. λz. s (n s z)
```

Is that the only possible one? Can you think of another one?

Another encoding of data types, called *Scott* encoding:

```
    c_0' = λs. λz. z
succ' n = λs. λz. s n
```

Notice the difference:

```
c_2' = succ' (succ' c_0')
     ≡  λs. λz. s (λs. λz. s (λs. λz. z))
```

Church encodes *folding*, while Scott encodes *pattern matching*.

Scott Encoding of Numerals

```
     c0' = λs. λz. z
succ' n = λs. λz. s n
```

Predecessor:

?

Scott Encoding of Numerals

```
     c₀' = λs. λz. z
succ' n = λs. λz. s n
```

Predecessor:

```
pred' n = n id c₀'
```

(where `id = λx. x`)

Scott Encoding of Numerals

```
     c₀' = λs. λz. z
succ' n = λs. λz. s n
```

Predecessor:

```
pred' n = n id c₀'
```

(where `id = λx. x`)

Addition:

?

Scott Encoding of Numerals

```
       c₀' = λs. λz. z
succ' n = λs. λz. s n
```

Predecessor:

```
pred' n = n id c₀'
```

(where `id = λx. x`)

Addition:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Any problems with this?

Scott Encoding of Numerals

```
     c0' = λs. λz. z
succ' n = λs. λz. s n
```

Predecessor:

```
pred' n = n id c0'
```

(where `id = λx. x`)

Addition:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Any problems with this?

This definition **refers to itself**! *Not a lambda term...*
We seem to need recursion...

Divergence and Recursion in the Lambda Calculus

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self =` λ`f. f f`

Seems a bit suspicious...

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self` = λ`f. f f`

Seems a bit suspicious...

Quizz: what's this? (recall: `double f x = f (f x)`)

```
self double
```

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self =` λ`f. f f`

Seems a bit suspicious...

Quizz: what's this? (recall: `double f x = f (f x)`)

```
self double
  ≡ double double
```

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self = λf. f f`

Seems a bit suspicious...

Quizz: what's this? (recall: `double f x = f (f x)`)

```
self double
  ≡ double double
  ≡ λx. double (double x)
  ≡ λx. λx'. (double x) ((double x) x')
  ≡ λx. λx'. double x (double x x')
  ≡ λx. λx'. x (x (x (x x')))
```

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self = λf. f f`

Seems a bit suspicious...

Quizz: what's this? (recall: `double f x = f (f x)`)

```
self double
  ≡ double double
  ≡ λx. double (double x)
  ≡ λx. λx'. (double x) ((double x) x')
  ≡ λx. λx'. double x (double x x')
  ≡ λx. λx'. x (x (x (x x')))
  ≡  ``quadruple''
```

Self Application

What can we say about the following definition? (self application)

```
self f = f f
```

i.e., `self` = λ`f. f f`

Seems a bit suspicious...

Quizz: what's this? (recall: `double f x = f (f x)`)

```
self double
  ≡ double double
  ≡ λx. double (double x)
  ≡ λx. λx'. (double x) ((double x) x')
  ≡ λx. λx'. double x (double x x')
  ≡ λx. λx'. x (x (x (x x')))
  ≡  ``quadruple''
```

Now how about this?

```
self self
```

Divergence in the Lambda Calculus

Self-applying self application... *what could go wrong?*

```
self self
```

Divergence in the Lambda Calculus

Self-applying self application... *what could go wrong?*

```
self self
  = (λf. f f) self
```

Divergence in the Lambda Calculus

Self-applying self application... *what could go wrong?*

```
self self
  = (λf. f f) self
  ≡ self self
  ≡ ...
```

`self self` is a term that reduces to itself in one step.

Within self-application great power lies.

Divergence in the Lambda Calculus

Self-applying self application... *what could go wrong?*

```
self self
  = (λf. f f) self
  ≡ self self
  ≡ ...
```

`self self` is a term that reduces to itself in one step.

Within self-application great power lies. — Yoda, probably

Can we harness this power?

Hacking self application

Recall our problem:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Let's rewrite `plus'` as a proper lambda term,
using *indirect recursion* by self application...

Hacking self application

Recall our problem:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Let's rewrite `plus'` as a proper lambda term,
using *indirect recursion* by self application...

Idea: take an argument that will hold the current definition itself!

```
mkPlus' myself n m =
  n (λpn. succ (myself myself pn m)) m
```

Hacking self application

Recall our problem:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Let's rewrite `plus'` as a proper lambda term,
using *indirect recursion* by self application...

Idea: take an argument that will hold the current definition itself!

```
mkPlus' myself n m =
  n (λpn. succ (myself myself pn m)) m

plus' = mkPlus' mkPlus'    ≡  self mkPlus'
```

Hacking self application

Recall our problem:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Let's rewrite `plus'` as a proper lambda term,
using *indirect recursion* by self application...

Idea: take an argument that will hold the current definition itself!

```
mkPlus' myself n m =
  n (λpn. succ (myself myself pn m)) m

plus' = mkPlus' mkPlus'     ≡  self mkPlus'

plus' n m = mkPlus' mkPlus' n m
  ≡ n (λpn. succ (mkPlus' mkPlus' pn m)) m
```

Hacking self application

Recall our problem:

```
plus’ n m = n (λpn. succ (plus’ pn m)) m
```

Let’s rewrite `plus’` as a proper lambda term,
using *indirect recursion* by self application...

Idea: take an argument that will hold the current definition itself!

```
mkPlus’ myself n m =
  n (λpn. succ (myself myself pn m)) m

plus’ = mkPlus’ mkPlus’     ≡  self mkPlus’

plus’ n m = mkPlus’ mkPlus’ n m
  ≡ n (λpn. succ (mkPlus’ mkPlus’ pn m)) m
  ≡ n (λpn. succ (plus’ pn m)) m
```

Mission accomplished!

Hacking self application

Recall our problem:

```
plus' n m = n (λpn. succ (plus' pn m)) m
```

Let's rewrite `plus'` as a proper lambda term,
using *indirect recursion* by self application...

Idea: take an argument that will hold the current definition itself!

```
mkPlus' myself n m =
  n (λpn. succ (myself myself pn m)) m

plus' = mkPlus' mkPlus'     ≡  self mkPlus'

plus' n m = mkPlus' mkPlus' n m
  ≡ n (λpn. succ (mkPlus' mkPlus' pn m)) m
  ≡ n (λpn. succ (plus' pn m)) m
```

Mission accomplished! *But we can do better (more convenient)...*

Divergence, more formally

Recursion and divergence are intertwined, so we need to consider divergent terms.

$$\texttt{omega} \quad = \quad \texttt{(}\lambda\texttt{x. x x) (}\lambda\texttt{x. x x)}$$

Note that `omega` evaluates in one step to itself!
So evaluation of `omega` never reaches a normal form: it *diverges*.

Divergence, more formally

Recursion and divergence are intertwined, so we need to consider divergent terms.

$$\texttt{omega} = (\lambda\texttt{x. x x})\ (\lambda\texttt{x. x x})$$

Note that `omega` evaluates in one step to itself!
So evaluation of `omega` never reaches a normal form: it *diverges*.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of `omega` that are *very* useful...

Recall: Normal forms

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

Recall: Normal forms

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

But are there any stuck terms in the pure λ-calculus?

Recall: Normal forms

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

But are there any stuck terms in the pure λ-calculus?

Yes. Example: x

Recall: Normal forms

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

But are there any stuck terms in the pure λ-calculus?

Yes. Example: x

BUT no stuck *closed* terms
(a closed term is a term without free variables)

Recall: Normal forms

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

But are there any stuck terms in the pure λ-calculus?

Yes. Example: x

BUT no stuck *closed* terms
(a closed term is a term without free variables)

Note: closedness is preserved by evaluation!

Closed terms in the pure λ calculus never "crash"...

Towards recursion: Iterated application

Suppose `f` is some λ-abstraction, and consider the following variant of `omega`:

$$\mathtt{Y}_f \quad = \quad (\lambda \mathtt{x.\ f\ (x\ x)})\ (\lambda \mathtt{x.\ f\ (x\ x)})$$

Towards recursion: Iterated application

Suppose `f` is some λ-abstraction, and consider the following variant of `omega`:

$$\texttt{Y}_f \quad = \quad (\lambda\texttt{x. f (x x)}) \; (\lambda\texttt{x. f (x x)})$$

Now the "pattern of divergence" becomes more interesting:

$$\texttt{Y}_f$$
$$=$$
$$\underline{(\lambda\texttt{x. f (x x)}) \; (\lambda\texttt{x. f (x x)})}$$
$$\longrightarrow$$
$$\texttt{f (}\underline{(\lambda\texttt{x. f (x x)}) \; (\lambda\texttt{x. f (x x)})}\texttt{)}$$
$$\longrightarrow$$
$$\texttt{f (f (}\underline{(\lambda\texttt{x. f (x x)}) \; (\lambda\texttt{x. f (x x)})}\texttt{))}$$
$$\longrightarrow$$
$$\texttt{f (f (f (}\underline{(\lambda\texttt{x. f (x x)}) \; (\lambda\texttt{x. f (x x)})}\texttt{)))}$$
$$\longrightarrow$$
$$\ldots$$

$\texttt{Y}_f$ is still not very useful, since (like `omega`), all it does is diverge.

Is there any way we could "slow it down"?

Delaying divergence

$$\texttt{poisonpill} \quad = \quad \lambda\texttt{y. omega}$$

Note that `poisonpill` is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

$$\underline{(\lambda\texttt{p. fst (pair p fls) tru) poisonpill}}$$
$$\longrightarrow$$
$$\texttt{fst (pair poisonpill fls) tru}$$
$$\longrightarrow^*$$
$$\underline{\texttt{poisonpill tru}}$$
$$\longrightarrow$$
$$\texttt{omega}$$
$$\longrightarrow$$
$$\dots$$

A delayed variant of omega

Here is a variant of `omega` in which the delay and divergence are a bit more tightly intertwined:

$$\texttt{omegav} \quad = \quad \lambda\texttt{y. } (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; \texttt{y}$$

Note that `omegav` is a normal form. However, if we apply it to any argument `v`, it diverges:

$$\begin{array}{c} \texttt{omegav v} \\ = \\ \underline{(\lambda\texttt{y. } (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; \texttt{y}) \; \texttt{v}} \\ \longrightarrow \\ \underline{(\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y}))} \; \texttt{v} \\ \longrightarrow \\ (\lambda\texttt{y. } (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; \texttt{y}) \; \texttt{v} \\ = \\ \texttt{omegav v} \end{array}$$

Another delayed variant

Suppose `f` is a function. Define

$$\mathtt{z}_f \quad = \quad \lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y}$$

This term combines the "added `f`" from $\mathtt{Y}_f$ with the "delayed divergence" of `omegav`.

If we now apply $\mathtt{z}_f$ to an argument `v`, something interesting happens:

$$
\begin{array}{c}
\mathtt{z}_f\ \mathtt{v} \\
= \\
\underline{(\lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y})\ \mathtt{v}} \\
\longrightarrow \\
\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))}\ \mathtt{v} \\
\longrightarrow \\
\mathtt{f}\ (\lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y})\ \mathtt{v} \\
= \\
\mathtt{f}\ \mathtt{z}_f\ \mathtt{v}
\end{array}
$$

Since $\mathtt{z}_f$ and `v` are both values, the next computation step will be the reduction of `f` $\mathtt{z}_f$ — that is, before we "diverge," `f` gets to do some computation.

Now we are getting somewhere.

Recursion

Let

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

`f` looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function `fct`, which is passed as a parameter.

N.b.: for brevity, this example uses "real" numbers and booleans, infix syntax, etc. It can easily be translated into the pure lambda calculus (using Church numerals, etc.).

We can use `z` to "tie the knot" in the definition of `f` and obtain a real recursive factorial function:

$$
\begin{array}{c}
\texttt{z}_f\ \texttt{3} \\
\longrightarrow^* \\
\texttt{f}\ \texttt{z}_f\ \texttt{3} \\
= \\
(\lambda\texttt{fct.}\ \lambda\texttt{n.}\ \texttt{...})\ \texttt{z}_f\ \texttt{3} \\
\longrightarrow\ \longrightarrow \\
\texttt{if 3=0 then 1 else 3 * (}\texttt{z}_f\ \texttt{(pred 3))} \\
\longrightarrow^* \\
\texttt{3 * (}\texttt{z}_f\ \texttt{(pred 3)))} \\
\longrightarrow \\
\texttt{3 * (}\texttt{z}_f\ \texttt{2)} \\
\longrightarrow^* \\
\texttt{3 * (f }\texttt{z}_f\ \texttt{2)} \\
\ldots
\end{array}
$$

A Generic $\mathtt{z}$

If we define

$$\mathtt{z} \quad = \quad \lambda \mathtt{f}.\ \mathtt{z}_f$$

i.e.,

$$\mathtt{z} \quad = \quad \lambda \mathtt{f}.\ \lambda \mathtt{y}.\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y}$$

then we can obtain the behavior of $\mathtt{z}_f$ for any $\mathtt{f}$ we like, simply by applying $\mathtt{z}$ to $\mathtt{f}$.

$$\mathtt{z}\ \mathtt{f} \quad \longrightarrow \quad \mathtt{z}_f$$

For example:

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

Technical Note

The term `z` here is essentially the same as the `fix` discussed the book.

```
z =
  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y

fix =
  λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

`z` is hopefully slightly easier to understand, since it has the property that `z f v` $\longrightarrow^*$ `f (z f) v`, which `fix` does not (quite) share.

Programming in the Lambda Calculus, Continued (Again)

Recall: Church Booleans

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

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

$$\texttt{tru}\ c_0\ \texttt{omega}\ ?$$

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

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

Not what we want!

A better way

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

$$\texttt{unit} \quad = \quad \lambda\texttt{x. x}$$

Use a "conditional function":

$$\texttt{test} \quad = \quad \lambda\texttt{b. }\lambda\texttt{t. }\lambda\texttt{f. b t f unit}$$

If $\texttt{tru}'$ is or behaves like $\texttt{tru}$, $\texttt{fls}'$ is or behaves like $\texttt{fls}$, and $\texttt{s}$ and $\texttt{t}$ are arbitrary terms then

$$\texttt{test tru}' \texttt{ (}\lambda\texttt{dummy. s) (}\lambda\texttt{dummy. t)} \longrightarrow^{*} \texttt{s}$$
$$\texttt{test fls}' \texttt{ (}\lambda\texttt{dummy. s) (}\lambda\texttt{dummy. t)} \longrightarrow^{*} \texttt{t}$$

Recall: The z Operator

In the previous part, we defined an operator $\texttt{z}$ that calculates the "fixed point" of a function it is applied to:

$$\texttt{z} \;=\; \lambda\texttt{f}.\ \lambda\texttt{y}.\ (\lambda\texttt{x}.\ \texttt{f}\ (\lambda\texttt{y}.\ \texttt{x}\ \texttt{x}\ \texttt{y}))\ (\lambda\texttt{x}.\ \texttt{f}\ (\lambda\texttt{y}.\ \texttt{x}\ \texttt{x}\ \texttt{y}))\ \texttt{y}$$

That is, if $\texttt{z}_f = \texttt{z f}$ then $\texttt{z}_f\ \texttt{v} \longrightarrow^* \texttt{f}\ \texttt{z}_f\ \texttt{v}$.

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

$\texttt{fact}\ \texttt{c}_3 \longrightarrow^*$

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:

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

Now:

$$\begin{array}{c}\texttt{realnat (times } \texttt{c}_2\ \texttt{c}_2\texttt{)} \\ \longrightarrow^* \\ \texttt{succ (succ (succ (succ zero)))}.\end{array}$$

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:

$$\texttt{whack (fact } c_3\texttt{)} \\ \longrightarrow^* \\ \lambda\texttt{s. }\lambda\texttt{z. s (s (s (s (s (s z)))))}$$

Equivalence of Lambda Terms

Recall: Church Numerals

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

$$
\begin{aligned}
c_0 &= \lambda s.\ \lambda z.\ z \\
c_1 &= \lambda s.\ \lambda z.\ s\ z \\
c_2 &= \lambda s.\ \lambda z.\ s\ (s\ z) \\
c_3 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z))
\end{aligned}
$$

Other lambda-terms represent common operations on numbers:

$$
\mathtt{scc} = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)
$$

Recall: Church Numerals

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

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$

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 $\texttt{scc}\ \texttt{c}_n$ evaluates to $\texttt{c}_{n+1}$.

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 $\texttt{scc}\ \texttt{c}_2$ behaves the same as $\texttt{c}_3$ when applied to two arguments.

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

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

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 $\mathtt{v}_1 \ldots \mathtt{v}_n$ such that one of

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

and

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

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`:

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

`poisonpill unit`

diverges

Proving behavioral inequivalence

Example:

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

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

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

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 $v_1 \ldots v_n$, 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:

```
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 up to one element (i.e., $n \leq 1$), note that both `tru` / `tru v1` and `tru'` / `tru' v1` 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 `tru v1 v2 v3 ... vn` and `tru' v1 v2 v3 ... vn` reduce to `v1 v3 ... vn`. 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

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

and

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

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