Formal Methods

Lecture 6

(B. Pierce's slides for the book "Types and Programming Languages")

This Saturday, 10 November 2018, room 335 (FSEGA), we will recover the following activities:

- **1 Formal Methods course, 7.30-9.30**
- **1 Programming Paradigms course, 9.30-11.30**
- **1 Formal Methods course, 11.30-13.30**

Programming in the Lambda-Calculus, Continued

Normal forms

Recall:

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

Recursion in the Lambda-Calculus

Divergence

```
omega  =  (λx. x x) (λx. x x)
```

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it *diverges*.

Divergence

```
omega  =  (λx. x x) (λx. x x)
```

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

Iterated Application

Suppose f is some λ-abstraction, and consider the following term:

$$Y_f \quad = \quad (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Now the “pattern of divergence” becomes more interesting:

$$\begin{array}{c}
Y_f \\
= \\
\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))} \\
\longrightarrow \\
f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}) \\
\longrightarrow \\
f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})) \\
\longrightarrow \\
f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))) \\
\longrightarrow \\
\ldots
\end{array}$$

Y_f is still not very useful, since (like `omega`), all it does is diverge.

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.

$$\underline{(\lambda\texttt{p. fst (pair p fls) tru) poisonpill}}$$

$$\longrightarrow$$

$$\texttt{fst (pair poisonpill fls) tru}$$

$$\longrightarrow^{*}$$

$$\underline{\texttt{poisonpill tru}}$$

$$\longrightarrow$$

$$\texttt{omega}$$

$$\longrightarrow$$

$$\ldots$$

A delayed variant of omega

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

```
omegav   =
λy. (λx. (λy. x x y)) (λx. (λy. x x y)) y
```

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

```
omegav v
=
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
−→
(λx. (λy. x x y)) (λx. (λy. x x y)) v
−→
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
=
omegav v
```

Another delayed variant

Suppose `f` is a function. Define

```
Zf    =  λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
```

This term combines the "added `f`" from Y_f with the "delayed divergence" of `omegav`.

If we now apply Z_f to an argument v, something interesting happens:

$$
\begin{gathered}
Z_f \ v \\
= \\
\cancel{(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v} \\
\longrightarrow \\
\underline{(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))}\ v \\
\longrightarrow \\
f\ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v \\
= \\
f\ Z_f\ v
\end{gathered}
$$

Since Z_f and v are both values, the next computation step will be the reduction of $f\ Z_f$ — 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:

```
Z_f 3
−→*
f Z_f 3
=
(λfct. λn. ...) Z_f 3
−→ −→
if 3=0 then 1 else 3 * (Z_f (pred 3))
−→*
3 * (Z_f (pred 3)))
−→
3 * (Z_f 2)
−→*
3 * (f Z_f 2)
...
```

A Generic Z

If we define

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

i.e.,

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

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` :

```
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.

Testing booleans

Recall:

```
tru  =  λt. λf. t
fls  =  λt. λf. f
```

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

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

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

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

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

A better way

A dummy "unit value," for forcing evaluation of thunks:

```
unit = λx. x
```

A "conditional function":

```
test = λb. λt. λf. b t f unit
```

If `b` is a boolean (i.e., it behaves like either `tru` or `fls`), then, for arbitrary *terms* `s` and `t`, either

$$\texttt{b (λdummy. s) (λdummy. t)} \longrightarrow^* \texttt{s}$$

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

$$\texttt{b (λdummy. s) (λdummy. t)} \longrightarrow^* \texttt{t}$$

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

The Z Operator

we defined an operator Z that calculates the “fixed point” of a function it is applied to:

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

That is, $\texttt{z f v} \xrightarrow{*} \texttt{f (z f) v}$.

Factorial

As an example, we defined the factorial function in lambda-calculus as follows:

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

For the sake of the example, we used “regular” booleans, numbers, etc.

This could be translated “straightforwardly” into the pure lambda-calculus.

Let’s do this.

Factorial

```
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 c0`?)

Factorial

A better version:

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

Displaying numbers

`fact c6` $\xrightarrow{*}$

Displaying numbers

```
fact c6    *
——→
    (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))
```

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:

$$\texttt{realnat (times c2 c2)} \xrightarrow{}^{*} \texttt{succ (succ (succ (succ zero)))}.$$

Displaying numbers

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

```
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)))))
```

A Larger Example

Head and tail functions for streams:

```
streamhd = λs. fst (s unit)
streamtl = λs. snd (s unit)
```

A stream of increasing numbers:

```
upfrom =
  fix
    (λr.
      λn.
        λdummy.
          pair n (r (scc n)))
```

Some tests:

```
whack (streamhd (upfrom c0))
    -->* c0

whack (streamhd (streamtl (upfrom c0)))
    -->* c2

whack (streamhd (streamtl (streamtl (upfrom c0))))
    -->* c4
```

Mapping over streams:

```
streammap =
  fix
    (λsm.
      λf.
        λs.
          λdummy.
            pair (f (streamhd s)) (sm f (streamtl s)))
```

Some tests:

```
evens = streammap double (upfrom c0);
whack (streamhd evens);
   /* yields c0 */
whack (streamhd (streamtl evens));
   /* yields c2 */
whack (streamhd (streamtl (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.

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

Representing Numbers

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... doesn't work

One possibility:

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

Unfortunately, this is false.
E.g.:

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

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.

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

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

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."

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

Aside:

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

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

$$s\ v_1\ v_2\ ...\ v_n$$

and

$$t\ v_1\ v_2\ ...\ v_n$$

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 $v_1 \ldots v_n$ 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.

Proving behavioral inequivalence

Example:

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

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

$$\texttt{poisonpill unit}$$
diverges

Proving behavioral inequivalence

Example:

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

```
tru (λx. x) poisonpill (λx. x)
    −→* (λx. x)(λx. x)
    −→* λx. x

fls (λx. x) poisonpill (λx. x)
−→* poisonpill (λx. x)
```

, which diverges

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 just one element (i.e., $n = 1$), note that both $\mathtt{tru}\ v_1$ and $\mathtt{tru'}\ v_1$ 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 $\mathtt{tru}\ v_1\ v_2\ v_3\ \ldots\ v_n$ and $\mathtt{tru'}\ v_1\ v_2\ v_3\ \ldots\ v_n$ 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 = (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 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:

$$FV\,(x) = \{\,x\}$$
$$FV\,(\lambda x.\ t_1) = FV\,(t_1) \setminus \{\,x\}$$
$$FV\,(t_1\ \ t_2) = FV\,(t_1) \cup FV\,(t_2)$$

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

$$size(x) = 1$$
$$size(\lambda x.\ t_1) = size(t_1) + 1$$
$$size(t_1\ \ t_2) = size(t_1) + size(t_2) + 1$$

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

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

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

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

Induction on derivations

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

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

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

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

Induction on derivations

Induction principle for the small-step evaluation relation.

To show that a property P holds for all derivations of $t \longrightarrow t'$, it suffices to show that

- P holds for all derivations that use the rule E-AppAbs;
- P holds for all derivations that end with a use of E-App1 assuming that P holds for all subderivations; and
- P holds for all derivations that end with a use of E-App2 assuming that 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 $t \longrightarrow t'$, that $FV(t) \supseteq FV(t')$.

There are three cases.

- If the derivation of $t \longrightarrow t'$ 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:

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

- 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.

More About Bound Variables

Substitution

Our definition of evaluation is based on the “substitution” of values for free variables within terms.

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

But what is substitution, exactly? How do we define it?

Substitution

For example, what does

```
(λx. x (λy. x y)) (λx. x y x)
```

reduce to?

Note that this example is not a "complete program" — the whole term is not closed. We are mostly interested in the reduction behavior of closed terms, but reduction of open terms is also important in some contexts:

- program optimization
- alternative reduction strategies such as "full beta-reduction"

Formalizing Substitution

Consider the following definition of substitution:

$$[x \to s]x = s$$
$$[x \to s]y = y \qquad \text{if } x \neq y$$
$$[x \to s](\lambda y.t_1) = \lambda y.\ ([x \to s]t_1)$$
$$[x \to s](t_1\ t_2) = ([x \to s]t_1)([x \to s]t_2)$$

What is wrong with this definition?

Formalizing Substitution

It substitutes for free and *bound* variables!

$$[x \rightarrow y](\lambda x.\ x) = \lambda x.y$$

This is not what we want!

Substitution, take two

$$[x \to s]x = s$$
$$[x \to s]y = y \qquad \text{if } x \neq y$$
$$[x \to s](\lambda y.t_1) = \lambda y.\ ([x \to s]t_1) \qquad \text{if } x \neq y$$
$$[x \to s](\lambda x.t_1) = \lambda x.\ t_1$$
$$[x \to s](t_1\ t_2) = ([x \to s]t_1)([x \to s]t_2)$$

What is wrong with this definition?

Substitution, take two

What is wrong with this definition? It suffers from *variable capture*!

$$[x \to y](\lambda y.x) = \lambda x.\ x$$

This is also not what we want.

Substitution, take three

$$
\begin{array}{ll}
[x \to s]x = s & \\
[x \to s]y = y & \text{if } x \neq y \\
[x \to s](\lambda y.t_1) = \lambda y.\ ([x \to s]t_1) & \text{if } x \neq y,\ y \notin FV(s) \\
[x \to s](\lambda x.t_1) = \lambda x.\ t_1 & \\
[x \to s](t_1\ t_2) = ([x \to s]t_1)([x \to s]t_2) &
\end{array}
$$

What is wrong with this definition?

Substitution, take three

What is wrong with this definition?

Now substition is a *partial function!*

E.g., $[x \to y](\lambda y.x)$ is undefined.

But we want an result for every substitution.

Bound variable names shouldn’t matter

It’s annoying that that the “spelling” of bound variable names is causing trouble with our definition of substitution.

Intuition tells us that there shouldn’t be a difference between the functions `λx.x` and `λy.y`. Both of these functions do exactly the same thing.

Because they differ only in the names of their bound variables, we’d like to think that these *are* the same function.

We call such terms *alpha-equivalent*.

Alpha-equivalence classes

In fact, we can create equivalence classes of terms that differ only in the names of bound variables.

When working with the lambda calculus, it is convenient to think about these *equivalence classes*, instead of raw terms.

For example, when we write $\lambda x.x$ we mean not just this term, but the class of terms that includes $\lambda y.y$ and $\lambda z.z$.

We can now freely choose a different *representative* from a term's alpha-equivalence class, whenever we need to, to avoid getting stuck.