CIS 500

Software Foundations

Fall 2005

19 September

Announcements

- Homework 1 was due at noon.
- Homework 2 is on the web page.

The Lambda Calculus

The lambda-calculus

- If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest interesting programming language...
  - Turing complete
  - higher order (functions as data)
  - main new feature: variable binding and lexical scope
- The e. coli of programming language research
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

```
plus3   =   λx. succ (succ (succ x))
```

This function exists independent of the name `plus3`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

```
plus3   =   λx. succ (succ (succ x))
```

This function exists independent of the name `plus3`.

On this view, `plus3 (succ 0)` is just a convenient shorthand for "the function that, given `x`, yields `succ (succ (succ x))`, applied to `succ 0`."

```
plus3 (succ 0)   =   (λx. succ (succ (succ x))) (succ 0)
```

Essentials

We have introduced two primitive syntactic forms:

- abstraction of a term `t` on some subterm `x`:

  `λx. t`

  "The function that, when applied to a value `v`, yields `t` with `v` in place of `x`."

- application of a function to an argument:

  $\mathtt{t}_1\ \mathtt{t}_2$

  "the function $\mathtt{t}_1$ applied to the argument $\mathtt{t}_2$"

Abstractions over Functions

Consider the **λ**-abstraction

```
g   =   λf. f (f (succ 0))
```

Note that the parameter variable `f` is used in the function position in the body of `g`. Terms like `g` are called higher-order functions.

If we apply `g` to an argument like `plus3`, the “substitution rule” yields a nontrivial computation:

```
g plus3  =    (λf. f (f (succ 0))) (λx. succ (succ (succ x)))
         i.e. (λx. succ (succ (succ x)))
                ((λx. succ (succ (succ x))) (succ 0))
         i.e. (λx. succ (succ (succ x)))
                (succ (succ (succ (succ 0))))
         i.e. succ (succ (succ (succ (succ (succ 0)))))
```

Abstractions Returning Functions

Consider the following variant of `g`:

```
double   =  λf. λy. f (f y)
```

I.e., `double` is the function that, when applied to a function `f`, yields a function that, when applied to an argument `y`, yields `f (f y)`.

Example

```
          double plus3 0
=         (λf. λy. f (f y))
            (λx. succ (succ (succ x)))
            0
i.e.      (λy. (λx. succ (succ (succ x)))
                    ((λx. succ (succ (succ x))) y))
            0
i.e.      (λx. succ (succ (succ x)))
                    ((λx. succ (succ (succ x))) 0)
i.e.      (λx. succ (succ (succ x)))
                    (succ (succ (succ 0)))
i.e.      succ (succ (succ (succ (succ (succ 0)))))
```

The Pure Lambda-Calculus

As the preceding examples suggest, once we have **λ**-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the "pure lambda-calculus"— everything is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function

Formalities

Syntax

`t ::=`	*terms*
`x`	*variable*
`λx.t`	*abstraction*
`t t`	*application*

Terminology:

- terms in the pure **λ**-calculus are often called **λ**-terms
- terms of the form `λx. t` are called **λ**-abstractions or just abstractions

Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left

  E.g., `t u v` means `(t u) v`, not `t (u v)`

- Bodies of λ- abstractions extend as far to the right as possible

  E.g., `λx. λy. x y` means `λx. (λy. x y)`, not `λx. (λy. x) y`

Scope

The λ-abstraction term `λx.t` binds the variable `x`.

The scope of this binding is the body `t`.

Occurrences of `x` inside `t` are said to be bound by the abstraction.

Occurrences of `x` that are not within the scope of an abstraction binding `x` are said to be free.

```
λx. λy. x y z
```

Scope

The `λ`-abstraction term `λx.t` binds the variable `x`.

The scope of this binding is the body `t`.

Occurrences of `x` inside `t` are said to be bound by the abstraction.

Occurrences of `x` that are not within the scope of an abstraction binding `x` are said to be free.

```
 λx. λy. x y z
λx. (λy. z y) y
```

Values

`v`	::=		*values*
		`λx.t`	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$ is “the term that results from substituting free occurrences of $\mathtt{x}$ in $\mathtt{t}_{12}$ with $\mathtt{v}_{12}$.”

Operational Semantics

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$ is “the term that results from substituting free occurrences of $\mathtt{x}$ in $\mathtt{t}_{12}$ with $\mathtt{v}_{12}$.”

Congruence rules:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

Terminology

A term of the form `(λx.t) v` — that is, a **λ**-abstraction applied to a value — is called a redex (short for “reducible expression”).

Programming in the Lambda-Calculus

Multiple arguments

Above, we wrote a function `double` that returns a function as an argument.

$$\texttt{double} \quad = \quad \lambda\texttt{f. }\lambda\texttt{y. f (f y)}$$

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.

In general, `λx. λy. t` is a function that, given a value `v` for `x`, yields a function that, given a value `u` for `y`, yields `t` with `v` in place of `x` and `u` in place of `y`.

That is, `λx. λy. t` is a two-argument function.

The “Church Booleans”

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

$$\begin{aligned} & \texttt{tru v w} & \\ = \quad & \underline{(\lambda\texttt{t}.\lambda\texttt{f}.\texttt{t})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow \quad & \underline{(\lambda\texttt{f}.\ \texttt{v})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow \quad & \texttt{v} & \text{reducing the underlined redex} \end{aligned}$$

$$\begin{aligned} & \texttt{fls v w} & \\ = \quad & \underline{(\lambda\texttt{t}.\lambda\texttt{f}.\texttt{f})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow \quad & \underline{(\lambda\texttt{f}.\ \texttt{f})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow \quad & \texttt{w} & \text{reducing the underlined redex} \end{aligned}$$

Functions on Booleans

$$\texttt{not} \quad = \quad \texttt{λb. b fls tru}$$

That is, `not` is a function that, given a boolean value `v`, returns `fls` if `v` is `tru` and `tru` if `v` is `fls`.

Functions on Booleans

$$\texttt{and} \quad = \quad \lambda\texttt{b.}\ \lambda\texttt{c.}\ \texttt{b c fls}$$

That is, `and` is a function that, given two boolean values `v` and `w`, returns `w` if `v` is `tru` and `fls` if `v` is `fls`

Thus `and v w` yields `tru` if both `v` and `w` are `tru` and `fls` if either `v` or `w` is `fls`.

Pairs

```
pair = λf.λs.λb. b f s
fst = λp. p tru
snd = λp. p fls
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.

By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean.

Example

$$
\begin{array}{lll}
 & \texttt{fst (pair v w)} & \\
= & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{f. }\lambda\texttt{s. }\lambda\texttt{b. b f s) v}}\texttt{ w)} & \text{by definition} \\
\longrightarrow & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{s. }\lambda\texttt{b. b v s) w}}\texttt{)} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{fst (}\lambda\texttt{b. b v w)} & \text{reducing the underlined redex} \\
= & \underline{\texttt{(}\lambda\texttt{p. p tru) (}\lambda\texttt{b. b v w)}} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(}\lambda\texttt{b. b v w) tru}} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{tru v w} & \text{reducing the underlined redex} \\
\longrightarrow^{*} & \texttt{v} & \text{as before.}
\end{array}
$$

Church numerals

Idea: represent the number n by a function that "repeats some action n times."

$$
\begin{aligned}
\texttt{c}_0 &= \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{z} \\
\texttt{c}_1 &= \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s z} \\
\texttt{c}_2 &= \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s (s z)} \\
\texttt{c}_3 &= \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s (s (s z))}
\end{aligned}
$$

That is, each number n is represented by a term $\texttt{c}_n$ that takes two arguments, `s` and `z` (for "successor" and "zero"), and applies `s`, n times, to `z`.

Functions on Church Numerals

Successor:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

Functions on Church Numerals

Successor:

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

Addition:

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

Multiplication:

$$\texttt{times = } \lambda\texttt{m. } \lambda\texttt{n. m (plus n) c}_0$$

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c₀
```

Zero test:

Functions on Church Numerals

Successor:

`scc = λn. λs. λz. s (n s z)`

Addition:

`plus = λm. λn. λs. λz. m s (n s z)`

Multiplication:

`times = λm. λn. m (plus n)` c_0

Zero test:

`iszro = λm. m (λx. fls) tru`

Functions on Church Numerals

Successor:

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

Addition:

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

Multiplication:

$$\texttt{times = }\lambda\texttt{m. }\lambda\texttt{n. m (plus n) c}_0$$

Zero test:

$$\texttt{iszro = }\lambda\texttt{m. m (}\lambda\texttt{x. fls) tru}$$

What about predecessor?

Predecessor

zz = pair c_0 c_0

ss = λp. pair (snd p) (scc (snd p))

Predecessor

zz = pair c_0 c_0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)

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.

Are there any stuck terms in the pure λ-calculus?

Prove it.

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.

Are there any stuck terms in the pure λ-calculus?

Prove it.

Does every term evaluate to a normal form?

Prove it.

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)
```

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...

Recursion in the Lambda Calculus

Iterated Application

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

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

Iterated Application

Suppose $\mathtt{f}$ is some λ-abstraction, and consider the following term:

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

Now the “pattern of divergence” becomes more interesting:

$$\begin{gathered}
\mathtt{Y_f} \\
= \\
\underline{\mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}} \\
\longrightarrow \\
\mathtt{f\ (}\underline{\mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}}\mathtt{)} \\
\longrightarrow \\
\mathtt{f\ (f\ (}\underline{\mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}}\mathtt{))} \\
\longrightarrow \\
\mathtt{f\ (f\ (f\ (}\underline{\mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}}\mathtt{)))} \\
\longrightarrow \\
\cdots
\end{gathered}$$

$\mathtt{Y_f}$ 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:

$$\texttt{omegav v}$$

$$=$$

$$\underline{(\lambda\texttt{y. } (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; \texttt{y}) \; \texttt{v}}$$

$$\longrightarrow$$

$$\underline{(\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y}))} \; \texttt{v}$$

$$\longrightarrow$$

$$(\lambda\texttt{y. } (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; (\lambda\texttt{x. } (\lambda\texttt{y. x x y})) \; \texttt{y}) \; \texttt{v}$$

$$=$$

$$\texttt{omegav v}$$

Another delayed variant

Suppose `f` is a function. Define

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

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 $\mathtt{v}$, something interesting happens:

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

Since $\mathtt{Z_f}$ and $\mathtt{v}$ are both values, the next computation step will be the reduction of $\mathtt{f\ \ Z_f}$ — that is, before we "diverge," $\mathtt{f}$ 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:

$$\mathtt{Z_f\ 3}$$

$$\longrightarrow^{*}$$

$$\mathtt{f\ Z_f\ 3}$$

$$=$$

$$\mathtt{(\lambda fct.\ \lambda n.\ \ldots)\ Z_f\ 3}$$

$$\longrightarrow\quad\longrightarrow$$

$$\mathtt{if\ 3{=}0\ then\ 1\ else\ 3\ *\ (Z_f\ (pred\ 3))}$$

$$\longrightarrow^{*}$$

$$\mathtt{3\ *\ (Z_f\ (pred\ 3)))}$$

$$\longrightarrow$$

$$\mathtt{3\ *\ (Z_f\ 2)}$$

$$\longrightarrow^{*}$$

$$\mathtt{3\ *\ (f\ Z_f\ 2)}$$

$$\ldots$$

A Generic $\mathtt{Z}$

If we define

$$\mathtt{Z} = \lambda\mathtt{f}.\ \mathtt{Z}_{\mathtt{f}}$$

i.e.,

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

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

$$\mathtt{Z}\ \mathtt{f} \longrightarrow \mathtt{Z}_{\mathtt{f}}$$

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.

Proofs about the Lambda Calculus

Two induction principles

Like before, we have mentioned two ways to prove that properties are true of the untyped lambda calculus.

- Structural induction
- Induction on derivation of `t` → `t'`.

Let's do an example of the latter.

Induction principle

Recall the induction principle for the small-step evaluation relation.

We can show a property $\boldsymbol{P}$ is true for all derivations of $\mathtt{t} \rightarrow \mathtt{t}'$, when

- $\boldsymbol{P}$ holds for all derivations that use the rule E-AppAbs.
- $\boldsymbol{P}$ holds for all derivations that end with a use of E-App1 assuming that $\boldsymbol{P}$ holds for all subderivations.
- $\boldsymbol{P}$ holds for all derivations that end with a use of E-App2 assuming that $\boldsymbol{P}$ holds for all subderivations.

Example

We can formally define the set of free variables in a $\boldsymbol{\lambda}$-term as follows:

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

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

Induction on derivation

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

We have three cases.

Induction on derivation

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

We have three cases.

- The derivation of $\mathtt{t} \longrightarrow \mathtt{t'}$ could just be a use of E-AppAbs. In this case, $\mathtt{t}$ is $(\lambda\mathtt{x.u})\mathtt{v}$ which steps to $[\mathtt{x}\mapsto\mathtt{v}]\mathtt{u}$.

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

◆ The derivation could end with a use of E-App1. In other words, we have a derivation of $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ and we use it to show that $\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2$.

By induction $\mathbf{FV}(\mathtt{t}_1) \supseteq \mathbf{FV}(\mathtt{t}_1')$.

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

- The derivation could end with a use of E-App1. In other words, we have a derivation of $\mathtt{t}_1 \rightarrow \mathtt{t}_1'$ and we use it to show that $\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}_1'\ \mathtt{t}_2$.

  By induction $\mathbf{FV}(\mathtt{t}_1) \supseteq \mathbf{FV}(\mathtt{t}_1')$.

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

- The derivation could end with a use of E-App2. Here, we have a derivation of $\mathtt{t}_2 \rightarrow \mathtt{t}_2'$ and we use it to show that $\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}_1\ \mathtt{t}_2'$. This case is analogous to the previous case.

More about bound variables

Substitution

Our definition of evaluation was based on the substitution of values for free variables within terms.

E-AppAbs

$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}$

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

Formalizing Substitution

Consider the following definition of substitution:

$$
\begin{aligned}
&[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} \\
&[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} && \text{if } \mathtt{x} \neq \mathtt{y} \\
&[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) \\
&[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2)
\end{aligned}
$$

What is wrong with this definition?

Formalizing Substitution

Consider the following definition of substitution:

$$\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}$$

What is wrong with this definition?

It substitutes for free and bound variables!

$[\mathtt{x} \mapsto \mathtt{y}](\lambda\mathtt{x}.\ \mathtt{x}) = \lambda\mathtt{x}.\mathtt{y}$

This is not what we want.

Substitution, take two

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{x}.\mathtt{t}_1) = \lambda\mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is wrong with this definition?

Substitution, take two

$$\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } x \neq y \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } x \neq y \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{x}.\mathtt{t}_1) = \lambda\mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}$$

What is wrong with this definition?

It suffers from variable capture!

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda\mathtt{y}.\mathtt{x}) = \lambda\mathtt{x}.\ \mathtt{x}$$

This is also not what we want.

Substitution, take three

$$
\begin{array}{ll}
[x \mapsto s]x = s & \\
[x \mapsto s]y = y & \text{if } x \text{ is not } y \\
[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1) & \text{if } x \neq y,\ y \notin \mathbf{FV}(s) \\
[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1 & \\
[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2) &
\end{array}
$$

What is wrong with this definition?

Substitution, take three

$$
\begin{array}{ll}
[x \mapsto s]x = s & \\
[x \mapsto s]y = y & \text{if } x \text{ is not } y \\
[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1) & \text{if } x \neq y,\ y \notin \mathbf{FV}(s) \\
[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1 & \\
[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2) &
\end{array}
$$

What is wrong with this definition?

Now substition is a partial function!

$[x \mapsto y](\lambda y.x)$ is undefined.

But we want an answer for every substitution.

Bound variable names shouldn't matter

It's annoying that that the names of bound variables are 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 will do 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 **`λx.x`** we mean not just this term, but the class of terms that includes **`λy.y`** and **`λz.z`**.

Unfortunately, we have to be more clever when implementing the lambda calculus in ML... (cf. TAPL chapters 6 and 7)

Substitution, for alpha-equivalence classes

Now consider substitution as an operation over alpha-equivalence classes of terms:

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

Examples:

- $[x \mapsto y](\lambda y.x)$ must give the same result as $[x \mapsto y](\lambda z.x)$. We know the latter is $\lambda z.y$, so that is what we will use for the former.
- $[x \mapsto y](\lambda x.z)$ must give the same result as $[x \mapsto y](\lambda w.z)$. We know the latter is $\lambda w.z$ so that is what we use for the former.