263-2200

Types and Programming Languages
Outline

The Untyped Lambda Calculus

Basics
   Syntax
   Operational Semantics
   Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
   Substitution

An Implementation in Prolog
Outline

The Untyped Lambda Calculus

Basics
- Syntax
- Operational Semantics
- Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
- Substitution

An Implementation in Prolog
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)
- Indeed, in the lambda-calculus, *all* computation happens by means of function *abstraction* and *application*.
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

"The lambda-calculus"
Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3} \ x = \text{succ} \ (\text{succ} \ (\text{succ} \ x)) \]

That is, “\text{plus3} \ x \ is \ \text{succ} \ (\text{succ} \ (\text{succ} \ x)).”
Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3 } x = \text{succ (succ (succ } x)) \]

That is, “\text{plus3 } x \text{ is succ (succ (succ } x)).”

Q: What is \text{plus3} itself?
Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3 } x = \text{succ} (\text{succ} (\text{succ } x)) \]

That is, “\text{plus3 } x \text{ is succ (succ (succ } x))\text{}.”

Q: What is \text{plus3} itself?

A: \text{plus3} is the function that, given \text{x}, yields \text{succ (succ (succ } x)).
Intuitions

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

\[ \text{plus3 } x = \text{succ } (\text{succ } (\text{succ } x)) \]

That is, “\text{plus3} x \text{ is } \text{succ } (\text{succ } (\text{succ } x)).”

Q: What is \text{plus3} itself?

A: \text{plus3} is the function that, given \( x \), yields \( \text{succ } (\text{succ } (\text{succ } x)) \).

\[ \text{plus3 } = \lambda x. \text{succ } (\text{succ } (\text{succ } x)) \]

This function exists independent of the name \text{plus3}.

\[ \lambda x. \text{succ } (\text{succ } (\text{succ } x)) \]
So \( \text{plus3} (\text{succ} \ 0) \) is just a convenient shorthand for “the function that, given \( x \), yields \( \text{succ} (\text{succ} (\text{succ} \ 0)) \), applied to \( \text{succ} \ 0 \).”

\[
\text{plus3} (\text{succ} \ 0) \\
= \\
(\lambda x. \text{succ} (\text{succ} (\text{succ} \ x))) (\text{succ} \ 0)
\]
Abstractions over Functions

Consider the $\lambda$-abstraction

$$g = \lambda f. \, f \,(f \,(\text{succ} \,0))$$

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 $\text{plus}3$, the “substitution rule” yields a nontrivial computation:

$$g \, \text{plus}3$$
$$\begin{align*}
&= \, (\lambda f. \, f \,(f \,(\text{succ} \,0))) \,(\lambda x. \, \text{succ} \,(\text{succ} \,(\text{succ} \, x))) \\
&= \, (\lambda x. \, \text{succ} \,(\text{succ} \,(\text{succ} \, x))) \\
&\quad \,(\,(\lambda x. \, \text{succ} \,(\text{succ} \,(\text{succ} \, x))) \,(\text{succ} \,0)) \\
&= \, (\lambda x. \, \text{succ} \,(\text{succ} \,(\text{succ} \, x))) \\
&\quad \,(\,\text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,0))))) \\
&= \, \text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,(\text{succ} \,0)))))))
\end{align*}$$
Consider the following variant of $g$:

$$\textit{double} = \lambda f. \lambda y. f (f \, y)$$

That is, \textit{double} is the function that, when applied to a function $f$, yields a \textit{function} that, when applied to an argument $y$, yields $f \, (f \, y)$. 


Example

double plus3 0

\[= \ (\lambda f. \ \lambda y. \ f \ (f \ y))\]
\[\hspace{1cm} (\lambda x. \ succ \ (succ \ (succ \ x)))\]
\[0\]
\[= \ (\lambda y. \ (\lambda x. \ succ \ (succ \ (succ \ x))))\]
\[\hspace{1cm} ((\lambda x. \ succ \ (succ \ (succ \ x)))) \ y)\]
\[0\]
\[= \ (\lambda x. \ succ \ (succ \ (succ \ x)))\]
\[\hspace{1cm} ((\lambda x. \ succ \ (succ \ (succ \ x))) \ 0)\]
\[= \ (\lambda x. \ succ \ (succ \ (succ \ x)))\]
\[\hspace{1cm} (\succ \ (\succ \ (\succ \ 0)))\]
\[= \ succ \ (\succ \ (\succ \ (\succ \ (\succ \ 0))))\]
As the preceding examples suggest, once we have $\lambda$-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
Outline

The Untyped Lambda Calculus

Basics

Syntax
Operational Semantics
Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities

Substitution

An Implementation in Prolog
Syntax

t ::= terms
  x variable
  \lambda x. t abstraction
  t t application

Terminology:
- terms in the pure \( \lambda \)-calculus are often called \( \lambda \)-terms
- terms of the form \( \lambda x. t \) are called \( \lambda \)-abstractions or just abstractions
Since $\lambda$-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 $\lambda$-abstractions extend as far to the right as possible**
  
  *E.g.*, $\lambda x. \lambda y. x\ y$ means $\lambda x. (\lambda y. x\ y)$, not $\lambda x. (\lambda y. x)\ y$
Scope

- The \( \lambda \)-abstraction term \( \lambda x. t \) binds the variable \( x \).
- The \textit{scope} of this binding is the \textit{body} \( t \).
- Occurrences of \( x \) inside \( t \) are said to be \textit{bound} by the abstraction.
- Occurrences of \( x \) that are \textit{not} within the scope of an abstraction binding \( x \) are said to be \textit{free}.

Test (which variable occurrences are bound and which are free):

\[
\lambda x. \lambda y. x \; y \; z
\]

\[
\lambda x. (\lambda y. z \; y) \; y
\]

- A term with no free variables is said to be \textit{closed}.  


17 / 93
Values

\[ \nu ::= \quad \text{values} \]

\[ \lambda x. t \quad \text{abstraction value} \]
Operational Semantics

Computation rule:

\[(\lambda x. \ t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12}\]  \hspace{1cm} \text{(E-AppAbs)}

Notation: \([x \mapsto v_2]t_{12}\) is “the term that results from substituting free occurrences of \(x\) in \(t_{12}\) with \(v_2\).”

Congruence rules:

\[
\begin{array}{c}
\text{Given: } t_1 \rightarrow t'_1 \\
\hline
\text{Then: } t_1 \ t_2 \rightarrow t'_1 \ t_2
\end{array}
\]  \hspace{1cm} \text{(E-App1)}

\[
\begin{array}{c}
\text{Given: } t_2 \rightarrow t'_2 \\
\hline
\text{Then: } v_1 \ t_2 \rightarrow v_1 \ t'_2
\end{array}
\]  \hspace{1cm} \text{(E-App2)}
A term of the form $(\lambda x.t_1) \ t_2$ — that is, a $\lambda$-abstraction applied to a term $t_2$ — is called a redex (short for “reducible expression”).

Note that the way we have defined evaluation, the right-hand term (e.g., $t_2$) of a redex must also be a value in order for a redex to be performed.
Overview of Alternative Evaluation Strategies

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus.

The evaluation strategy we have chosen — call-by-value — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call-by-name (cf. Haskell)
- Normal-order (leftmost/outermost)
- Full (non-deterministic) beta-reduction
Let us consider the reduction of the following term:

$$(\lambda x. x) ((\lambda x. x) (\lambda z. (\lambda x. x) z))$$

which we can write more readably as:

$$id_1 (id_2 (\lambda z. id_3 z))$$
Full Beta-Reduction

Any redex may be reduced at any time. Thus, one permissible reduction order is:

\[ id_1 (id_2 (\lambda z. id_3 z)) \]
\[ \rightarrow id_1 (id_2 (\lambda z. z)) \]
\[ \rightarrow id_1 (\lambda z. z) \]
\[ \rightarrow \lambda z. z \]
\[ \not\rightarrow \]
Normal Order

The *leftmost-outermost* redex is always selected for reduction. Note that reduction descends into the bodies of $\lambda$-abstractions.

\[
\begin{align*}
\text{id}_1 \ (\text{id}_2 \ (\lambda z. \ \text{id}_3 \ z)) & \rightarrow \text{id}_2 \ (\lambda z. \ \text{id}_3 \ z) \\
& \rightarrow \lambda z. \ \text{id}_3 \ z \\
& \rightarrow \lambda z. \ z \\
& \rightarrow \\
\end{align*}
\]
Call-By-Name (non-strict)

Essentially, \textit{leftmost-outermost} with the following restriction:

- Reduction is \textit{not} performed within the body of a \(\lambda\)-abstraction.

\[
\begin{align*}
\text{id}_1 \ (\text{id}_2 \ (\lambda z. \text{id}_3 \ z)) & \rightarrow \text{id}_2 \ (\lambda z. \text{id}_3 \ z) \\
& \rightarrow \lambda z. \text{id}_3 \ z \\
& \not\rightarrow
\end{align*}
\]
Call-By-Value (strict)

Essentially, *leftmost-outermost* with the following restrictions:

- The *right-hand* side of a redex must be reduced to a value before the redex is performed.
- Reduction is *not* performed within the body of a \( \lambda \)-abstraction.

\[
\begin{align*}
\text{id}_1 \ (\text{id}_2 \ (\lambda z. \text{id}_3 \ z)) & \\
\rightarrow & \ \text{id}_1 \ (\lambda z. \text{id}_3 \ z) \\
\rightarrow & \ \lambda z. \text{id}_3 \ z
\end{align*}
\]
Outline

The Untyped Lambda Calculus

Basics
  Syntax
  Operational Semantics
  Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
  Substitution

An Implementation in Prolog
Multiple arguments

Consider the function *double*, which takes a function as an argument and returns a function as its result.

\[
double = \lambda f. \lambda y. f (f \ y)
\]

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

In general, \(\lambda x. \lambda y. t\) 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 \(y\).

In essence, \(\lambda x. \lambda y. t\) can be seen as a *two-argument* function.
The “Church Booleans”

\[ \text{tru} \ = \ \lambda t. \ \lambda f. \ t \]
\[ \text{fls} \ = \ \lambda t. \ \lambda f. \ f \]

\[ \text{tru} \ v \ w \]
\[ = \ (\lambda t. \lambda f. \ t) \ v \ w \quad \text{by definition} \]
\[ \rightarrow \ (\lambda f. \ v) \ w \quad \text{reducing the underlined redex} \]
\[ \rightarrow \ v \quad \text{reducing the underlined redex} \]

\[ \text{fls} \ v \ w \]
\[ = \ (\lambda t. \lambda f. \ f) \ v \ w \quad \text{by definition} \]
\[ \rightarrow \ (\lambda f. \ f) \ w \quad \text{reducing the underlined redex} \]
\[ \rightarrow \ w \quad \text{reducing the underlined redex} \]
Functions on Booleans

\[ \text{not} = \lambda b. \ b \ fls \ tru \]

That is, \textit{not} is a function that, given a boolean value \( v \), returns \textit{fls} if \( v \) is \textit{tru} and \textit{tru} if \( v \) is \textit{fls}. 
Functions on Booleans

\[ and = \lambda b. \lambda c. b \, c \, fls \]

That is, \textit{and} is a function that, given two boolean values \( v \) and \( w \), returns \( w \) if \( v \) is \textit{tru} and \textit{fls} if \( v \) is \textit{fls}.

Thus, \textit{and} \( v \, w \) yields \textit{tru} if both \( v \) and \( w \) are \textit{tru} and \textit{fls} if either \( v \) or \( w \) is \textit{fls}.
Pairs

\[
\text{pair} = \lambda f. \lambda s. \lambda b. b \ f \ s \\
\text{fst} = \lambda p. p \ \text{tru} \\
\text{snd} = \lambda p. p \ \text{fls}
\]

That is, \text{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 \text{tru} and \( w \) if \( b \) is \text{fls}, so the first and second projection functions \text{fst} and \text{snd} can be implemented simply by supplying the appropriate boolean.
Example

\[
\text{fst } (\text{pair } v \ w)
\]
\[
= \text{fst } ((\lambda f. \lambda s. \lambda b. b \ f \ s) \ v \ w) \quad \text{by definition}
\]
\[
\longrightarrow \text{fst } ((\lambda s. \lambda b. b \ v \ s) \ w) \quad \text{reducing}
\]
\[
\longrightarrow \text{fst } (\lambda b. b \ v \ w) \quad \text{reducing}
\]
\[
= (\lambda p. p \ \text{tru}) \ (\lambda b. b \ v \ w) \quad \text{by definition}
\]
\[
\longrightarrow (\lambda b. b \ v \ w) \ \text{tru} \quad \text{reducing}
\]
\[
\longrightarrow \ \text{tru } v \ w \quad \text{reducing}
\]
\[
\longrightarrow^* v \quad \text{as before}
\]
Church numerals

Idea: represent the number $n$ by a function that “repeats some action $n$ times.”

\[
\begin{align*}
c_0 & = \lambda s. \lambda z. z \\
c_1 & = \lambda s. \lambda z. s \, z \\
c_2 & = \lambda s. \lambda z. s \, (s \, z) \\
c_3 & = \lambda s. \lambda z. s \, (s \, (s \, z))
\end{align*}
\]

That is, each number $n$ is represented by a term $c_n$ 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:

\[ scc = \lambda n. \lambda s. \lambda z. s (n s z) \]
Functions on Church Numerals

Successor:

\[ scc = \lambda n. \lambda s. \lambda z. s (n s z) \]

Addition:
Functions on Church Numerals

Successor:

\[
\text{scc} = \lambda n. \lambda s. \lambda z. s (n s z)
\]

Addition:

\[
\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)
\]
Functions on Church Numerals

Successor:

\[ \text{scc} = \lambda n. \lambda s. \lambda z. (n \ s \ z) \]

Addition:

\[ \text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. (m \ s \ (n \ s \ z)) \]

Multiplication:
Functions on Church Numerals

Successor:

\[ \text{successor} = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \]

Addition:

\[ \text{addition} = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z) \]

Multiplication:

\[ \text{multiplication} = \lambda m. \lambda n. m \ (\text{addition} \ n) \ c_0 \]
Functions on Church Numerals

Successor:

\[ \text{succ} = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \]

Addition:

\[ \text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z) \]

Multiplication:

\[ \text{times} = \lambda m. \lambda n. m \ (\text{plus} \ n) \ z_0 \]

Zero test:

\[ \text{iszzero} = \lambda m. m \ (\lambda x. \text{false}) \ \text{true} \]
Successor:

\[ scc = \lambda n. \lambda s. \lambda z. s (n s z) \]

Addition:

\[ plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z) \]

Multiplication:

\[ times = \lambda m. \lambda n. m (plus n) c_0 \]

Zero test:

\[ iszro = \lambda m. m (\lambda x. fls) tru \]
Functions on Church Numerals

Successor:

\[ scc = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \]

Addition:

\[ plus = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z) \]

Multiplication:

\[ times = \lambda m. \lambda n. m \ (plus \ n) \ c_0 \]

Zero test:

\[ iszro = \lambda m. m \ (\lambda x. \ fls) \ tru \]

What about predecessor?
Predecessor

\[ zz = \text{pair } c_0 \ c_0 \]

\[ ss = \lambda p. \text{pair } (\text{snd } p) \ (\text{scc } (\text{snd } p)) \]

\[ prd = \lambda m. \text{fst } (m \ ss \ zz) \]
Normal forms

Recall:

- A \textit{normal form} is a term that cannot take an evaluation step.
- A \textit{stuck} term is a normal form that is not a value.

Are there any stuck terms in the pure $\lambda$-calculus?
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 $\lambda$-calculus?

Does every term evaluate to a normal form?
Divergence

\[
omega = (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)
\]

- Note that \textit{omega} evaluates in one step to itself!

- So evaluation of \textit{omega} never reaches a normal form: it \textit{diverges}.
Divergence

\[ \omega = (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \]

- Note that \( \omega \) evaluates in one step to itself!

- So evaluation of \( \omega \) never reaches a normal form: it \textit{diverges}.

- Being able to write a divergent computation does not seem very useful in itself. However, there are variants of \( \omega \) that are \textit{very} useful...
Outline

The Untyped Lambda Calculus

Basics
  Syntax
  Operational Semantics
  Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
  Substitution

An Implementation in Prolog
Suppose $f$ is some $\lambda$-abstraction, and consider the following term:

$$Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$$
Iterated Application

Suppose $f$ is some $\lambda$-abstraction, and consider the following term:

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

Now the “pattern of divergence” becomes more interesting:

$$Y_f 
\rightarrow f ((\lambda x. f (x x)) (\lambda x. f (x x)))
\rightarrow f (f ((\lambda x. f (x x)) (\lambda x. f (x x))))
\rightarrow f (f (f ((\lambda x. f (x x)) (\lambda x. f (x x)))))
\rightarrow \cdots$$
$Y_f$ is still not very useful, since (like $\omega$), all it does is diverge.

Is there any way we could “slow it down”?
Delaying divergence

\[\text{poisonpill} = \lambda y. \text{omega}\]

Note that \textit{poisonpill} is a \textit{value} — 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.

\[
(\lambda p. \text{fst (pair } p \text{ fls) tru}) \text{poisonpill}
\]

\[\rightarrow\]

\[
\text{fst (pair poisonpill fls) tru}
\]

\[\rightarrow^*\]

\[
\text{poisonpill tru}
\]

\[\rightarrow\]

\[
\text{omega}
\]

\[\rightarrow\]

\[
\ldots
\]
A delayed variant of omega

Here is a variant of \( \text{omega} \) in which the delay and divergence are a bit more tightly intertwined:

\[
\omega' = \lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y
\]

Note that \( \omega' \) is a normal form. However, if we apply it to any argument \( v \), it diverges:

\[
\omega' v = (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \\
\quad \quad \quad \quad \rightarrow \quad (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) v \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad = \omega' v
\]
Another delayed variant

Suppose $f$ is a function. Define

$$Z_f = \lambda y. (\lambda x. f (\lambda y. x \times y)) (\lambda x. f (\lambda y. x \times y)) y$$

This term combines the “added $f$” from $Y_f$ with the “delayed divergence” of $\text{omegav}$. 
If we now apply \( Z_f \) to an argument \( v \), something interesting happens:

\[
Z_f \ v \\
= \\
(\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) \ v \\
\rightarrow \\
(\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) \ v \\
\rightarrow \\
f (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) \ v \\
= \\
f Z_f \ v
\]

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

\[
    f = \lambda fct. \\
    \quad \lambda n. \\
    \quad \quad \text{if } n = 0 \text{ then } 1 \\
    \quad \quad \text{else } n \times (fct (\text{pred } n))
\]

The function \( f \) looks just like the ordinary factorial function, except that, in place of a recursive call in the last line, 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:

\[
Z_f 3 \\
\quad \rightarrow^* \\
f \ Z_f \ 3 \\
= \\
(\lambda fct. \ \lambda n. \ldots) \ Z_f \ 3 \\
\quad \rightarrow \quad \rightarrow \\
if \ 3 = 0 \ then \ 1 \ else \ 3 \times (Z_f \ (pred \ 3)) \\
\quad \rightarrow^* \\
3 \times (Z_f \ (pred \ 3)) \\
\quad \rightarrow^* \\
3 \times (f \ Z_f \ 2) \\
\quad \rightarrow^* \\
3 \times (f \ Z_f \ 2) \\
\ldots
\]
If we define

\[ Z = \lambda f. Z_f \]
\[ = \lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y \]

then we can obtain the behavior of \( Z_f \) for any \( f \) we like, simply by applying \( Z \) to \( f \).

\[ Z f \rightarrow Z_f \]
Example

\[
\text{fact} = Z (\lambda fct. \\
\quad \lambda n. \\
\quad \quad \text{if } n = 0 \text{ then } 1 \\
\quad \quad \text{else } n \ast (fct (\text{pred } n)) \\
\quad )
\]
The term \( Z \) here is essentially the same as the \( \text{fix} \) discussed the book.

\[
Z = \lambda f. \lambda y. (\lambda x. f (\lambda x. x x y)) (\lambda x. f (\lambda y. x x y))
\]

\[
\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))
\]

\( Z \) is hopefully slightly easier to understand, since it has the property that \( Z \ f \ v \longrightarrow^* \ f \ (Z \ f) \ v \), which \( \text{fix} \) does not (quite) share.
Outline

The Untyped Lambda Calculus

Basics
  Syntax
  Operational Semantics
  Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
  Substitution

An Implementation in Prolog
An example of structural induction on terms

Define the set of free variables in a lambda-term as follows:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(\lambda x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1 t_2) &= FV(t_1) \cup FV(t_2)
\end{align*}
\]

Define the size of a lambda-term as follows:

\[
\begin{align*}
\text{size}(x) &= 1 \\
\text{size}(\lambda x.t_1) &= 1 + \text{size}(t_1) \\
\text{size}(t_1 t_2) &= 1 + \text{size}(t_1) + \text{size}(t_2)
\end{align*}
\]

**Theorem:** \(|FV(t)| \leq \text{size}(t)|\).
An example of structural induction on terms

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

Proof: By induction on the structure of $t$.

- If $t$ is a variable, then $|FV(t)| = 1 = \text{size}(t)$.
- If $t$ is an abstraction $\lambda x. t_1$, then

\[
\begin{align*}
|FV(t)| \\
= & \ |FV(t_1) \setminus \{x\}| & \text{by defn} \\
\leq & \ |FV(t_1)| & \text{by arithmetic} \\
\leq & \ \text{size}(t_1) & \text{by induction hypothesis} \\
\leq & \ 1 + \text{size}(t_1) & \text{by arithmetic} \\
= & \ \text{size}(t) & \text{by defn}.
\end{align*}
\]
Theorem: $|FV(t)| \leq \text{size}(t)$.

Proof: By induction on the structure of $t$.

- If $t$ is an application $t_1 \ t_2$, then

  \[
  |FV(t)| \\
  = |FV(t_1) \cup FV(t_2)| \quad \text{by defn} \\
  \leq |FV(t_1)| + |FV(t_2)| \quad \text{by arithmetic} \\
  \leq \text{size}(t_1) + \text{size}(t_2) \quad \text{by IH and arithmetic} \\
  \leq 1 + \text{size}(t_1) + \text{size}(t_2) \quad \text{by arithmetic} \\
  = \text{size}(t) \quad \text{by defn}.
  \]
Induction on Derivations

Theorem: if \( t \rightarrow t' \) then \( FV(t) \supseteq FV(t') \).
Induction on derivations

We must prove, for all derivations of $t \rightarrow t'$, that

\[ FV(t) \supseteq FV(t') \]

There are three cases.
Induction on derivations

We must prove, for all derivations of $t \rightarrow t'$, that

$$FV(t) \supseteq FV(t')$$

There are three cases.

- If the derivation of $t \rightarrow t'$ is a use of E-AppAbs, then $t$ is $(\lambda x. t_1) v$ and $t'$ is $[x \mapsto v] t_1$. Reason as follows:

$$FV(t) = FV((\lambda x. t_1) v)$$
$$= FV(t_1) \setminus \{x\} \cup FV(v)$$
$$\supseteq FV([x \mapsto v] t_1)$$
$$= FV(t')$$
If the derivation ends with a use of E-App1, then $t$ has the form $t_1 \ t_2$ and $t'$ has the form $t'_1 \ t_2$, and we have a subderivation of $t_1 \rightarrow t'_1$

By the induction hypothesis, $FV(t_1) \supseteq FV(t'_1)$. Now calculate:

\[
FV(t) = FV(t_1 \ t_2) \\
= FV(t_1) \cup FV(t_2) \\
\supseteq FV(t'_1) \cup FV(t_2) \\
= FV(t'_1 \ t_2) \\
= FV(t')
\]
If the derivation ends with a use of E-App1, then $t$ has the form $t_1 \ t_2$ and $t'$ has the form $t'_1 \ t_2$, and we have a subderivation of $t_1 \rightarrow t'_1$

By the induction hypothesis, $FV(t_1) \supseteq FV(t'_1)$. Now calculate:

$$FV(t) = FV(t_1 \ t_2)$$
$$= FV(t_1) \cup FV(t_2)$$
$$\supseteq FV(t'_1) \cup FV(t_2)$$
$$= FV(t'_1 \ t_2)$$
$$= FV(t')$$

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

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

\[(\lambda x. \ t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12}\]  

(E-AppAbs)

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

For example, what does

$$(\lambda x. x (\lambda y. x y)) (\lambda x. x y x)$$

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:

\[
\begin{align*}
[x \mapsto s] x &= s \\
[x \mapsto s] y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y.([x \mapsto s] t_1) \\
[x \mapsto s](t_1 t_2) &= ([x \mapsto s] t_1)([x \mapsto s] t_2)
\end{align*}
\]

What is wrong with this definition?
Formalizing Substitution

Consider the following definition of substitution:

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y.([x \mapsto s] t_1) \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s] t_1) ([x \mapsto s] t_2)
\end{align*}
\]

What is wrong with this definition?

It substitutes for free and \textit{bound} variables!

\[
[x \mapsto y](\lambda x. x) = \lambda x. y
\]

This is not what we want!
Substitution, take two

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.([x \mapsto s]t_1) & \text{if } x \neq y \\
[x \mapsto s](t_1 t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?
Substitution, take two

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y.([x \mapsto s]t_1) & \text{if } x \neq y \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?

It suffers from \textit{variable capture}!

\[
[x \mapsto y](\lambda y. x) = \lambda y. y
\]

This is also not what we want.
Substitution, take three

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y.([x \mapsto s]t_1) & \text{if } x \neq y \land y \not\in FV(s) \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?
Substitution, take three

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y.([x \mapsto s]t_1) & \text{if } x \neq y \land y \notin \text{FV}(s) \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?

Now substitution is a \textit{partial function}!

\[E.g., \ [x \mapsto y](\lambda y. x) \text{ is undefined.}\]

But we want a result for every substitution.
Bound variable names shouldn’t matter

It’s annoying 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 $\lambda x.x$ and $\lambda y.y$. 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.
Substitution, for alpha-equivalence classes

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

\[
[x \mapsto s]x = s \\
[x \mapsto s]y = y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) = \lambda y. ([x \mapsto s]t_1) \quad \text{if } x \neq y \land y \notin \text{FV}(s) \\
[x \mapsto s](\lambda x. t_1) = \lambda x. t_1 \\
[x \mapsto s](t_1 \ t_2) = ([x \mapsto s]t_1) ([x \mapsto s]t_2)
\]

Examples:

1. \([x \mapsto y](\lambda y. x)\) 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.

2. \([x \mapsto y](\lambda x. z)\) 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.
So what does

\[(\lambda x. \ x \ (\lambda y. \ x \ y)) \ (\lambda x. \ x \ y \ x)\]

reduce to?
Remarks

- Bound variables can be *uniquely renamed* just prior to substitution in order to simulate alpha-conversion.

\[ [x \mapsto s](\lambda y.t_1) \]
\[ \Rightarrow \]
\[ [x \mapsto s](\lambda y_{\text{unique}}.t_1') \text{ where } t_1' \text{ is adjusted accordingly} \]

Remark

A very efficient variation of this idea is the *nameless term* (also sometimes called *de Bruijn term*). The details of nameless terms is described in Chapter 6.
Outline

The Untyped Lambda Calculus

Basics
  Syntax
  Operational Semantics
  Alternative Evaluation Strategies

Programming in the Lambda-Calculus

Recursion in the Lambda-Calculus

Formalities
  Substitution

An Implementation in Prolog
**Step 1: Design a Mapping to a Term Algebra**

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\text{map} \rightarrow \text{var}(x)$</td>
</tr>
<tr>
<td>$\lambda x.t$</td>
<td>$\text{map} \rightarrow \text{fn}(\text{var}(x), t)$</td>
</tr>
<tr>
<td>$t_1 ; t_2$</td>
<td>$\text{map} \rightarrow \text{app}(t_1, t_2)$</td>
</tr>
</tbody>
</table>
# Concrete Examples

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\text{map } \Rightarrow \text{var}(x)$</td>
</tr>
<tr>
<td>$\lambda x.x$</td>
<td>$\text{map } \Rightarrow \text{fn}(\text{var}(x), \text{var}(x))$</td>
</tr>
<tr>
<td>$(\lambda x.x) (\lambda x.x)$</td>
<td>$\text{map } \Rightarrow \text{app}(\text{fn}(\text{var}(x), \text{var}(x)), \text{fn}(\text{var}(y), \text{var}(y)))$</td>
</tr>
</tbody>
</table>
Step 2: Congruence Rules

\[
\begin{align*}
  t_1 & \rightarrow t_1' \\
  t_1 & t_2 \rightarrow t_1' t_2
\end{align*}
\]  
(E-App1)

reduce(STATE_IN, app( T1, T2 ), app( T1_1, T2 ), STATE_OUT )
:- not(isValue( T1)),
reduce(STATE_IN, T1, T1_1, STATE_OUT).

\[
\begin{align*}
  t_2 & \rightarrow t_2' \\
  v_1 & t_2 \rightarrow v_1 t_2'
\end{align*}
\]  
(E-App2)

reduce(STATE_IN, app( V1, T2 ), app( V1, T2_2 ), STATE_OUT )
:- isValue(V1),
not(isValue( T2)),
reduce(STATE_IN, T2, T2_2, STATE_OUT).
Step 2: Computation Rule

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

reduce(STATE_IN, app(fn(Var,Body_1), V2), Body_2, STATE_OUT )
  :-  isValue(V2),
      subst(STATE_IN, [Var,V2],Body_1,Body_2, STATE_OUT).
Step 2: Support Predicates

subst(STATE_IN, [var(X), V2], var(X), V2, STATE_IN).

subst(STATE_IN, [var(X), _], var(Y), var(Y), STATE_IN) :- not(X = Y).

subst(STATE_IN, [Var1, _], fn(Var1, Body), fn(Var1, Body), STATE_IN).

subst(STATE_IN, [Var1, V2], fn(Var2, Body2), fn(Var3, Body4), STATE_OUT) :- not(Var1 = Var2),
% Alpha-conversion assures that not(Var3 in FV(V2))
alphaConvert(STATE_IN, fn(Var2, Body2), fn(Var3, Body3), STATE_1),
subst(STATE_1, [Var1, V2], Body3, Body4, STATE_OUT).

subst(STATE_IN, [Var1, V2], app(T1, T2), app(T1_1, T2_2), STATE_OUT) :- subst(STATE_IN, [Var1, V2], T1, T1_1, STATE_1),
 subst(STATE_1, [Var1, V2], T2, T2_2, STATE_OUT).
alphaConvert( STATE_IN, fn(Var1, Body1), fn(Var2, Body2), STATE_OUT )
:- newVar(STATE_IN, Var2, STATE_OUT),
renameFree([Var1, Var2], Body1, Body2).

newVar( state(counter(X1)), var(X1), state(counter(X2)) )
:- X2 is X1 + 1.
renameFree( [Var1, Var2], Var1, Var2 ).

renameFree( [Var1, _], fn(Var1, Body), fn(Var1, Body) ).

renameFree( [Var1, Var2], fn(Var3, Body1), fn(Var3, Body2) )
  :- not(Var1 = Var3),
     renameFree(Var1, Var2, Body1, Body2).

renameFree( [Var1, Var2], app(T1, T2), app(T1_1, T2_2) )
  :- renameFree(Var1, Var2, T1, T1_1),
     renameFree(Var1, Var2, T2, T2_2).
Step 3: Syntactic Category Check(s)

isValue( fn(_,_) ).
Step 4: Implement a Computation Engine

% STATE0 = state(counter(0))
% \( T_0 \rightarrow T_1 \rightarrow T_2 \ldots \rightarrow T_{\text{normal form}} \)
% General Case:
eval(STATE0, T0, T_normal) :- reduce(STATE0, T0, T1, STATE1),
eval(STATE1, T1, T_normal).

% Base Case 1:
eval(\_, V, V) :- isValue(V).

% Base Case 2: this engine assumes “stuck” terms are possible
eval(STATE, T, stuck(T)) :- not(isValue(T)),
not(reduce(STATE, T, \_, \_)).