The Untyped Lambda Calculus

Ralf Lämmel
What’s the lambda calculus?

- It is the core of functional languages.
- ...
What’s the lambda calculus?

- It is the core of functional languages.
- It is a mathematical system for studying programming languages.
  - design, specification, implementation, type systems, et al.
- It comes in variations of typing: implicit/explicit/none.
- Formal systems built on top of simply typed lambda calculus:
  - System F — for studying polymorphism
  - System F <: — for studying subtyping
  - ...

© Ralf Lämmel, 2009-2012 unless noted otherwise
Language constructs

• Abstract syntax:

\[ M ::= x \mid M \, M \mid \lambda \, x.M \]

• \( M \) is a lambda term.

• An infinite set of variables \( x, y, z, \ldots \) is assumed.

• \( M \, N \) is an application.

  Function \( M \) is applied to the argument \( N \).

• \( \lambda \, x.M \) is an abstraction.

  The resulting function maps \( x \) to \( M \).

Lambda functions are anonymous.
• Church’s thesis:

**All intuitively computable functions are \( \lambda \)-definable.**

• An established equivalence of notions of computability:

**Set of Lambda-definable functions**

\[ = \] **Set of Turing-computable functions**
Syntax and semantics

• Syntax

\[ t ::= x \]
\[ \lambda x.t \]
\[ t t \]

\[ v ::= \lambda x.t \]

Terms

A term that cannot be reduced further.

• Evaluation

\[
\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}
\]
\[
\frac{t \to t'}{v t \to v t'}
\]

\[
(\lambda x.t) v \to [v/x] t
\]

Values (normal forms)

Reduce function position, then reduce argument position, then apply.
Syntactic sugar and conventions

- $M \ N_1 \ldots N_k$ means $\ldots (((M \ N_1) \ N_2) \ldots) \ N_k$.
  
  Function application groups from left to right.

- $\lambda x. x \ y$ means $(\lambda x. (x \ y))$.
  
  Function application has higher precedence.

- $\lambda x_1 x_2 \ldots x_k. M$ means $\lambda x_1. (\lambda x_2. (\ldots(\lambda x_k. (M)) \ldots)))$. 

This slide is derived from Jaakko Järvi’s slides for his course "Programming Languages", CPSC 604 @ TAMU.
Variable binding

• $\lambda$ is a binding operator:
  It binds a variable in the scope of the lambda abstraction.

• Examples:
  ✦ $\lambda x. M$ \hspace{1em} $x$ is bound (in the lambda abstraction)
  ✦ $\lambda x. x \ y$ \hspace{1em} $y$ is not bound (in the lambda abstraction).

• If a variable occurs in an expression without being bound, then it is called a **free** occurrence, or a free variable. Other occurrences of variables are called **bound**.

• A **closed term** is one without free variable occurrences.
Variable binding — precise definition

\[ FV(M) \] defines the set of free variables in the term \( M \)

\[
FV(x) = \{x\}
\]

\[
FV(MN) = FV(M) \cup FV(N)
\]

\[
FV(\lambda x. M) = FV(M) \setminus \{x\}
\]
Exercise: what are the free and bound variable occurrences in these terms?

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

\(\lambda x. (\lambda y. x \ y) y\)
Substitution and $\beta$-equivalence

- **Computation for the $\lambda$-calculus is based on substitution.**
- Substitution is defined by the equational axiom:
  $$ (\lambda x. M) N = [N/x]M $$
- Think of substitution as invoking a function:
  - $(\lambda x. M)$ is the function,
  - $N$ is the argument,
  - Substitution takes care of parameter passing.

The terms on both sides are also called $\beta$-equivalent.

"$\rightarrow$" direction = $\beta$-reduction

Redex

This slide is derived from Jaakko Järvi’s slides for his course "Programming Languages", CPSC 604 @ TAMU.
\(\alpha\)-equivalence and conversion

- Names of bound variable are insignificant.

\(\lambda x.x\) defines the same function as \(\lambda y.y\)

- Suppose two terms differ only on the names of bound variables.

Then, they are said to be \(\alpha\)-equivalent (\(=\alpha\)).

- Equational axiom:

\[\lambda x.M = \lambda y.[y/x]M\]

where \(y\) does not appear in \(M\)

and substitution applies to free occurrences only.

Performing such renaming is also called \(\alpha\)-conversion.
Reduction $M \rightarrow N$

• Computation ($\rightarrow$) with the lambda calculus is then a series of
  
  ✦ $\beta$-reductions, and
  
  ✦ (“implicit”) $\alpha$-conversions.

• Reflexive, transitive closure

• $M \rightarrow^* N$ means $M$ reduces to $N$ in zero or more steps.
**Inductive definition of substitution**

\[
\begin{align*}
[N/x]x &= N \\
[N/x]y &= y, \text{ } y \text{ any variable different from } x \\
[N/x](M_1 \ M_2) &= ([N/x]M_1) ([N/x]M_2) \\
[N/x](\lambda x. M) &= \lambda x. M \\
[N/x](\lambda y. M) &= \lambda y.([N/x]M), \text{ } y \text{ not free in } N
\end{align*}
\]

**Examples**

\[
\begin{align*}
[z/x]x &= z \\
[z/x](\lambda x. xx) &= \lambda x. xx \\
[z/x](\lambda y. y x) &= \lambda y. \lambda y. y z \\
[z/x](\lambda z. x z) &= \lambda a. \lambda z. a a
\end{align*}
\]

If this condition is not met, then alpha conversion is needed.
Properties of reduction (i.e., semantics)

- How do we select redexes for reduction steps?
- Does the result depend on such a choice?
- Does reduction ultimately terminate with a normal form?
Illustration of different reductions
(We assume natural numbers with “+”.)

**Option 1**

\[
(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) \ 2
\]

\[
\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) \ x)) \ 2
\]

\[
\rightarrow (\lambda x. (\lambda y. y + 1) (x + 1)) \ 2
\]

\[
\rightarrow (\lambda x. (x+1+1)) \ 2
\]

\[
\rightarrow (2+1+1)
\]

\[
\rightarrow 4
\]

**Option 2**

\[
(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) \ 2
\]

\[
\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) \ x)) \ 2
\]

\[
\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) \ x)) \ 2
\]

\[
\rightarrow (\lambda y. y + 1) ((\lambda y. y + 1) \ 2)
\]

\[
\rightarrow ...
\]

\[
\rightarrow ...
\]

\[
\rightarrow 4
\]
Confluence

• Confluence: evaluation strategy is not significant for final value.

• That is: there is (at most) one normal form of a given expression.

Confluence

• $M \rightarrow^* N$ means $M$ reduces to $N$ in zero or more steps.

• Confluence

  If $M \rightarrow^* N$ and $M \rightarrow^* N'$,

  then there exists some $P$

  such that $N \rightarrow^* P$ and $N' \rightarrow^* P$. 
Strong normalization property of a calculus with reduction

• Definition:
  
  For every term $M$ there is a normal form $N$ such that $M \rightarrow^* N$.

• Strong normalization properties for lambda calculi:
  
  ✦ Untyped lambda calculus: no
  
  ✦ Simply-typed lambda calculus: yes
Evaluation/reduction strategies

✦ **Full beta reduction**
Reduce anywhere.

✦ **Applicative order** ("reduce the leftmost innermost redex")
Reduce argument before applying function.

✦ **Normal order** ("reduce the leftmost outermost redex")
Apply function before reducing argument.

Choice of strategy may impact termination behavior

- **eager** (strict)
- **lazy** (non-strict)

Extension vs. encoding

• Typical extensions
  (giving rise to so-called applied lambda calculi)
  ✦ Primitive types (numbers, Booleans, ...)
  ✦ Type constructors (tuples, records, ...)
  ✦ **Recursive functions**
  ✦ Effects (cell, exceptions, ...)
  ✦ ...

• Many extensions can be encoded in theory in terms of pure lambda calculus, except that such encoding is somewhat tedious.
Church Booleans

- Encodings of literals
  - true = \( \lambda t. \lambda f.t \)
  - false = \( \lambda t. \lambda f.f \)

- Conditional expression (if)
  - Expectations
    - test \( b \) \( v \) \( w \) \( \rightarrow^* \) \( v \), if \( b = \text{true} \)
    - test \( b \) \( v \) \( w \) \( \rightarrow^* \) \( w \), if \( b = \text{false} \)
  - Encoding
    - test = \( \lambda l. \lambda m. \lambda n. l \) \( m \) \( n \) = \( \lambda l. (\lambda m. (\lambda n. ((l m) n))) \)
Example reduction

\[(\lambda l.\lambda m.\lambda n.l\ m\ n)\ true\ v\ w\]

\[\rightarrow (\lambda m.\lambda n.true\ m\ n)\ v\ w\]

\[\rightarrow (\lambda n.true\ v\ n)\ w\]

\[\rightarrow true\ v\ w\]

\[\rightarrow (\lambda t.\lambda f.t)\ v\ w\]

\[\rightarrow (\lambda f.v)w\]

\[\rightarrow v\]
Church pairs

- A Boolean value picks either the 1st or the 2nd value of the pair.
- Construction and projections
  - $\text{pair} = \lambda f.\lambda s.\lambda b.b \ f \ s$
  - $\text{first} = \lambda p.p \ \text{true}$
  - $\text{second} = \lambda p.p \ \text{false}$
Church numerals

• Encodings of numbers
  ✦ \( c0 = \lambda s.\lambda z.z \)
  ✦ \( c1 = \lambda s.\lambda z.s\ z \)
  ✦ \( c2 = \lambda s.\lambda z.s\ (s\ z) \)
  ✦ \( c3 = \lambda s.\lambda z.s\ (s\ (s\ z)) \)
  ✦ ...

• Encodings of functions on numbers
  ✦ \( \text{succ} = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z) \)
  ✦ \( \text{plus} = \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \)
  ✦ \( \text{times} = \lambda m.\lambda n.m\ (\text{plus}\ n)\ c0 \)
  ✦ ...

A numeral \( n \) is a lambda abstraction that is parameterized by a case for zero, and a case for succ. In the body, the latter is applied \( n \) times to the former. This caters for primitive recursion.
Recursive functions

• Let us define the factorial function.

• Suppose we had “recursive function definitions”.

\[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n*f(n - 1) \]

• Let us do such recursion with anonymous functions.

• Fixed point combinators to the rescue!
Fixed points

- Consider a function $f : X \rightarrow X$.
- A fixed point of $f$ is a value $z$ such that $z = f(z)$.
- A function may have none, one, or multiple fixed points.
- Examples (functions and sets of fixed points):
  - $f(x) = 2x$ \quad \{0\}
  - $f(x) = x$ \quad \{0, 1, \ldots\}
  - $f(x) = x + 1$ \quad \emptyset
Fixed point combinators

We call $Y$ a fixed-point combinator if it satisfies the following *definitional property*:

For all $f : X \to X$ it holds that $Y f = f (Y f)$
Defining factorial as a fixed point

• Start from a recursive definition.

\[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n \ast f(n - 1) \]

• Eliminate self-reference; receive function as argument.

\[ g \equiv \lambda h. \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n \ast h(n - 1) \]

★ \( g \) takes a function (\( h \)) and returns a function.

• Define \( f \) as a fixed point.

\[ f \equiv Y \; g \]
Fixed points cont’d

Exercise for you!

• For example, apply definitional property to factorial g:

\[(Y \ g) \ 2\]

\[= [Y \ \text{def. prop.}] \quad (Y \ g) \ 2\]

\[= [\text{unfold } g] \quad \lambda \ h. \lambda \ n. \ \text{if } n == 0 \ \text{then} \ 1 \ \text{else} \ n*h(n - 1)) \ (Y \ g) \ 2\]

\[= [\beta \ \text{reduce}] \quad (\lambda \ n. \ \text{if } n == 0 \ \text{then} \ 1 \ \text{else} \ n*((Y \ g)(n - 1))) \ 2\]

\[= [\text{"-" reduce}] \quad \text{if } 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y \ g)(2 - 1))\]

\[= [\text{"if" reduce}] \quad \text{if } 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y \ g)(1))\]

\[= \ ...\]

\[= \ 2\]

This is as if we had extended evaluation.

Apply these steps one more time.
A lambda term for $\mathbf{Y}$

- One option:
  $$\mathbf{Y} = \lambda f . (\lambda x. f (x \ x))(\lambda x. f (x \ x))$$

- Verification of the definitional property:
  $$\mathbf{Y} \ g = g (\mathbf{Y} \ g)$$

- Proof:
  $$\begin{align*}
  \mathbf{Y} \ g \\
  &= [\text{unfold } \mathbf{Y}] (\lambda f . (\lambda x. f (x \ x))(\lambda x. f (x \ x))) \ g \\
  &= [\text{beta reduce}] (\lambda x. g(x \ x))(\lambda x. g(x \ x)) \\
  &= [\text{beta reduce}] \ g ((\lambda x. g(x \ x))(\lambda x. g(x \ x))) \\
  &= [\text{fold } \mathbf{Y}] \ g (\mathbf{Y} \ g)
  \end{align*}$$

Not suitable for applicative order.

Exercise for you!
Prolog as a sandbox for lambda calculi
The untyped lambda calculus

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/lambda/
Formalization of the lambda calculus

• Syntax

\[ t ::= x \]
\[ \lambda x. t \]
\[ t \; t \]

\[ v ::= \lambda x. t \]

Terms

Values (normal forms)

• Evaluation

\[
\frac{t \rightarrow t'}{t_1 \; t_2 \rightarrow t_1' \; t_2'} \]
\[
\frac{t \rightarrow t'}{v \; t \rightarrow v \; t'} \]

\[ (\lambda x. t) \; v \rightarrow [v/x]t \]}
Syntax of the untyped lambda calculus

term(var(X)) :- variable(X).
term(app(T1,T2)) :- term(T1), term(T2).
term(lam(X,T)) :- variable(X), term(T).

value(lam(X,T)) :- variable(X), term(T).

variable(X) :- atom(X).

Variables are Prolog atoms.
We illustrate Church Booleans and numerals. That is, we use a conditional (TEST) to select either C0 or C1.
Evaluation rules of
the untyped lambda calculus

```
&eval(app(T1,T2),app(T3,T2)) :-
  eval(T1,T3).
&eval(app(V,T1),app(V,T2)) :-
  value(V),
  eval(T1,T2).
&eval(app(lam(X,T1),V),T2) :-
  value(V),
  substitute(V,X,T1,T2).
```

Substitution (as needed for beta reduction) is the interesting part—both in the formal setting, and in Prolog.
Substitution

\[ [N/x]x = N \]
\[ [N/x]y = y, \text{ } y \text{ any variable different from } x \]
\[ [N/x](M_1 M_2) = ([N/x]M_1) ([N/x]M_2) \]
\[ [N/x](\lambda x. M) = \lambda x. M \]
\[ [N/x](\lambda y. M) = \lambda y.([N/x]M), \text{ } y \text{ not free in } N \]

**FV(M)** defines the set of free variables in the term **M**

\[ \text{FV}(x) = \{x\} \]
\[ \text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N) \]
\[ \text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\} \]
Substitution 1/3

```prolog
substitute(N,X,var(X),N).

substitute(_,X,var(Y),var(Y)) :- \+ X == Y.

substitute(N,X,app(M1,M2),app(M3,M4)) :-
    substitute(N,X,M1,M3),
    substitute(N,X,M2,M4).

substitute(_,X,lam(X,M),lam(X,M)).
```

The simple cases
substitute(N,X,lam(Y,M1),lam(Y,M2))
:-
  \+ X == Y,
  freevars(N,Xs),
  \+ member(Y,Xs),
  substitute(N,X,M1,M2).

Push down substitution into the body of the lambda abstraction if its bound variable \( Y \) does not occur freely in the target expression \( N \).
substitute(N,X,lam(Y,M1),lam(Z,M3))
:-
  \+ X =\= Y,
  freevars(N,Xs),
  member(Y,Xs),
  freshvar(Xs,Z),
  substitute(var(Z),Y,M1,M2),
  substitute(N,X,M2,M3).

If Y occurs freely in N, then we need to perform alpha conversion for Y. Hence, we find a fresh variable and convert the body M1 before we continue with the original substitution.
Free variables

freshvar(Xs,X)
  :-
      freshvar(Xs,X,0).

freshvar(Xs,N,N)
  :-
      \+ member(N,Xs).

freshvar(Xs,X,N1)
  :-
      member(N1,Xs),
      N2 is N1 + 1,
      freshvar(Xs,X,N2).

We use numbers as generated variables. We find the smallest number X that is not in Xs.
An applied, untyped lambda calculus

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/lambda/
We merge the syntax of NB and lambda calculus. In this manner, we get an applied lambda calculus (with all the applied bits of NB).
\( \lambda: \text{sample term} \)

\[
\text{app(app(}
\quad \text{% Twice function}
\text{lам(f,lам(x,app(вар(f),app(вар(f),вар(x))))))),}
\quad \text{% Increment function}
\text{lам(x,суч(var(x))))),}
\quad \text{% 2}
\text{суч(суч(зера))).}
\]

evaluates to 4
We construct the recursive `iseven` function and apply it to 3.

```prolog
app(app(
  % CBV fixed point combinator
  lam(f,app(
    lam(x,app(var(f),lam(y,app(app(var(x),var(x)),var(y))))),
    lam(x,app(var(f),lam(y,app(app(var(x),var(x)),var(y)))))),

  % iseven
  lam(e,lam(x,if(
    iszero(var(x)),
    true,
    if(
      iszero(pred(var(x))),
      false,
      app(var(e),pred(pred(var(x))))))))),

  % Argument to be tested
  succ(succ(succ(zero))))
).
```
Evaluation rules
the applied, untyped lambda calculus

Essentially, we merge the evaluation rules for NB and the untyped lambda calculus. However, we also need to upgrade substitution to cope with NB’s construct.

:- multifile eval/2.
:- multifile substitute/4.
:- multifile freevars/2.
:- ['../untyped/eval.pro'].
:- ['../..../nb/untyped/eval.pro'].
:- ['substitute.pro'].
:- ['freevars.pro'].
Substitution for applied part

substitute(_,_,true,true).
substitute(_,_,false,false).
substitute(_,_,zero,zero).
substitute(N,X,succ(T1),succ(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,pred(T1),pred(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,iszero(T1),iszero(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,if(T1a,T2a,T3a),if(T1b,T2b,T3b)) :-
    substitute(N,X,T1a,T1b),
    substitute(N,X,T2a,T2b),
    substitute(N,X,T3a,T3b).

This is all trivial code. We simply push substitution into NB’s terms.
Free variables for applied part

```prolog
freevars(true,[]).
freevars(false,[]).
freevars(zero,[]).
freevars(succ(T),FV) :- freevars(T,FV).
freevars(pred(T),FV) :- freevars(T,FV).
freevars(iszero(T),FV) :- freevars(T,FV).
freevars(if(T1,T2,T3),FV) :-
    freevars(T1,FV1),
    freevars(T2,FV2),
    freevars(T3,FV3),
    union(FV1,FV2,FV12),
    union(FV12,FV3,FV).
```

This is all trivial code. We simply traverse (and union) over NB’s terms.
• **Summary:** The untyped lambda calculus
  ✦ A concise core of functional programming.
  ✦ A foundation of computability.
  ✦ A Prolog model is again straightforward.

• **Prepping:** “Types and Programming Languages”
  ✦ Chapter 5

• **Outlook:**
  ✦ The simply-typed lambda calculus