Principles of Programming Languages

Lecture 04 Lambda Calculus

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

University of Freiburg, Germany

thiemann@informatik.uni-freiburg.de

07 May 2018

UNI FREIBURG

Plan

Lambda Calculus

- Foundational core calculus
- Basis for functional programming
- Turing complete
- Due to Alonzo Church (1936)

Plan

1. Interlude: Lambda Calculus
   - Syntax and Semantics
   - Programming in the Lambda Calculus
   - Evaluation Strategies
   - Applied Lambda Calculus

Syntax of Lambda Calculus

$$
\begin{aligned}
e ::= {} & x && \text{variable} \\
\mid {} & (\lambda x.e) && \text{(lambda) abstraction} \\
\mid {} & (e\ e) && \text{(function) application}
\end{aligned}
$$

- $x \in \mathrm{Var}$ a denumerable set

Remarks on Syntax

- Application is left-associative.

$$e_1\ e_2\ e_3 \equiv ((e_1\ e_2)\ e_3)$$

- The body of an abstraction reaches as far to the right as possible.

$$\lambda x\,.\,e_1\ e_2 \equiv (\lambda x\,.\,(e_1\ e_2))$$

- $\lambda xy\,.\,e$ stands for $\lambda x\,.\,\lambda y\,.\,e$ (analogously for more arguments).

Definition: Free and Bound Variables

The functions $FV(), BV() : \text{Exp} \to \mathcal{P}(\text{Var})$ return the set of *free* or *bound* variables of a lambda term, respectively.

$$
\begin{array}{rcl}
FV(x) & := & \{x\} \\
FV(e_0\ e_1) & := & FV(e_0) \cup FV(e_1) \\
FV(\lambda x\,.\,e) & := & FV(e) \setminus \{x\}
\end{array}
$$

$$
\begin{array}{rcl}
BV(x) & := & \emptyset \\
BV(e_0\ e_1) & := & BV(e_0) \cup BV(e_1) \\
BV(\lambda x\,.\,e) & := & BV(e) \cup \{x\}
\end{array}
$$

Furthermore, $Var(e) := FV(e) \cup BV(e)$ is the *set of variables* of e. A lambda term e is *closed* (e is a *combinator*) iff $FV(e) = \emptyset$.

Examples: Free and Bound Variables

$$
\begin{aligned}
FV(\lambda x\,.\,x) &= \emptyset \\
BV(\lambda x\,.\,x) &= \{x\} \\
FV(\lambda x\,.\,y) &= \{y\} \\
FV((\lambda x\,.\,x)\ y) &= \{y\} \\
BV((\lambda x\,.\,x)\ y) &= \{x\} \\
BV((\lambda x\,.\,x)\ x) &= \{x\} \\
FV((\lambda x\,.\,x)\ x) &= \{x\}
\end{aligned}
$$

!

Reduction Semantics

Reduction relation $e \longrightarrow e$

Beta
$$(\lambda x.e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]$$

CongLam
$$\frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'}$$

CongAppL
$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

CongAppR
$$\frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}$$

- Beta relies on **substitution** $e_1[x \mapsto e_2]$:
  "*substitute e_2 for x in e_1*"
- Substitution is tricky: it must not destroy lexical scope

Substitution: What can go wrong

- respect binding: $(\lambda x\,.\,x)[x \mapsto f] = (\lambda x\,.\,x)$

Substitution: What can go wrong

- respect binding: $(\lambda x\,.\,x)[x \mapsto f] = (\lambda x\,.\,x)$
- avoid capture: $(\lambda x\,.\,y)[y \mapsto x]$

Substitution: What can go wrong

- respect binding: $(\lambda x\,.\,x)[x \mapsto f] = (\lambda x\,.\,x)$
- avoid capture: $(\lambda x\,.\,y)[y \mapsto x]$
  - $= \lambda x\,.\,x$ would be WRONG

Substitution: What can go wrong

- respect binding: $(\lambda x\,.\,x)[x \mapsto f] = (\lambda x\,.\,x)$
- avoid capture: $(\lambda x\,.\,y)[y \mapsto x]$
  - $= \lambda x\,.\,x$ would be WRONG
  - $= \lambda x'\,.\,x$ is correct

Substitution: What can go wrong

- respect binding: $(\lambda x \,.\, x)[x \mapsto f] = (\lambda x \,.\, x)$
- avoid capture: $(\lambda x \,.\, y)[y \mapsto x]$
  - $= \lambda x \,.\, x$ would be WRONG
  - $= \lambda x' \,.\, x$ is correct
- must happen generally for $(\lambda x \,.\, e)[y \mapsto f]$ if $x \neq y$ and $x \in FV(f)$

Definition: Capture Avoiding Substitution

For $e, f \in E$, define $e[x' \mapsto f]$ inductively by:

$$
\begin{aligned}
x[x' \mapsto f] &= \begin{cases} f & \text{if } x = x' \\ x & \text{if } x \ne x' \end{cases} \\
(\lambda x\,.\,e)[x' \mapsto f] &= \begin{cases} \lambda x\,.\,e & \text{if } x = x' \\ \lambda x''\,.\,(e[x \mapsto x''][x' \mapsto f]) & \text{if } x \ne x', x'' \notin FV(e) \cup FV(f) \cup \{x'\} \end{cases} \\
(e_0\ e_1)[x' \mapsto f] &= (e_0[x' \mapsto f])\ (e_1[x' \mapsto f])
\end{aligned}
$$

Further Reduction Rules

Reduction Relation

Alpha

$$\lambda x\,.\,e \longrightarrow \lambda y.e[x \mapsto y] \quad y \notin FV(e)$$

Eta

$$(\lambda x\,.\,e\ x) \longrightarrow e \quad x \notin FV(e)$$

Remarks

- Alpha conversion is often used implicitly to keep free and bound variables apart
- Eta reduction is rarely used to describe execution
- Left hand side of a rule is called **redex**

More Relations Based on Reduction

Multi-step reduction aka reflexive transitive closure

$$e \xrightarrow{*} e \qquad \frac{e \longrightarrow e' \qquad e' \xrightarrow{*} e''}{e \xrightarrow{*} e''}$$

Equality aka symmetric reflexive transitive closure

$$\frac{e \longrightarrow e'}{e \longleftrightarrow e'} \qquad \frac{e' \longrightarrow e}{e \longleftrightarrow e'}$$

$$e \overset{*}{\longleftrightarrow} e \qquad \frac{{\color{red} e \longleftrightarrow e'} \qquad e' \overset{*}{\longleftrightarrow} e''}{e \overset{*}{\longleftrightarrow} e''}$$

More Remarks

- A β-reduction step corresponds closely to the intuitive notion of function application.
- Lambda terms will be considered equivalent if only the names of their bound variables differ (i.e., if they are α-convertible).

Let e be a lambda term. A lambda term e' is a **normal form** of e iff $e \xrightarrow{*}_\beta e'$ and if there is no e'' with $e' \longrightarrow_\beta e''$.

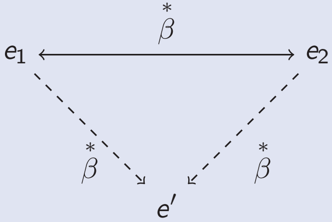
Remarks on Normal Forms

- Lambda terms with equivalent (equal up to α reduction) normal forms exhibit the same behavior.
- Some lambda terms do not have a normal form:

$$(\lambda x.x\ x)(\lambda x.x\ x) \longrightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

The Church-Rosser Theorem

Beta reduction has the **Church-Rosser property**:



In words: For all lambda terms e_1, e_2 with $e_1 \overset{*}{\leftrightarrow}_\beta e_2$, there is a lambda term e' with $e_1 \overset{*}{\rightarrow}_\beta e'$ and $e_2 \overset{*}{\rightarrow}_\beta e'$.

Corollary: Uniqueness of Normal Form

A lambda term e has at most one normal form up to Alpha reduction.

Plan

1. Interlude: Lambda Calculus
   - Syntax and Semantics
   - Programming in the Lambda Calculus
   - Evaluation Strategies
   - Applied Lambda Calculus

Desiderata

At first glance the lambda calculus lacks fundamental ingredients of a programming language:

- booleans and conditional,
- pairs / tuples / records,
- numbers, and
- recursion / while.

But all of them can be programmed, which makes lamdba calculus Turing equivalent.

Booleans and Conditionals

- Conditionals have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2.

Booleans and Conditionals

- Conditionals have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2.
- In lambda calculus booleans have an "active" interpretation that **performs** the selection by itself.

Booleans and Conditionals

- Conditionals have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2.
- In lambda calculus booleans have an "active" interpretation that **performs** the selection by itself.
- Thus, *true* is a lambda term that selects the first of two arguments, and *false* is one that selects the second:

$$\begin{aligned} \mathit{true} &= \lambda xy.x \\ \mathit{false} &= \lambda xy.y \end{aligned}$$

Booleans and Conditionals

- Conditionals have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2.
- In lambda calculus booleans have an "active" interpretation that **performs** the selection by itself.
- Thus, *true* is a lambda term that selects the first of two arguments, and *false* is one that selects the second:

$$\begin{array}{rcl} \mathit{true} & = & \lambda xy.x \\ \mathit{false} & = & \lambda xy.y \end{array}$$

- The conditional is the identity:

$$\mathit{ite} = \lambda bxy.b\ x\ y$$

Example Conditional

$$
\begin{aligned}
\mathit{if}\ \mathit{true}\ e_1\ e_2 &= (\lambda bxy.b\ x\ y)\ \mathit{true}\ e_1\ e_2 \\
&\rightarrow_\beta (\lambda xy.\mathit{true}\ x\ y)\ e_1\ e_2 \\
&\rightarrow_\beta^2 \mathit{true}\ e_1\ e_2 \\
&= (\lambda xy.x)\ e_1\ e_2 \\
&\rightarrow_\beta (\lambda y.e_1)\ e_2 \\
&\rightarrow_\beta e_1
\end{aligned}
$$

Numbers

Natural numbers can be represented by **Church numerals**. The Church numeral $\lceil n \rceil$ of a natural number n is a function that takes two parameters, a function f and some x, and applies f n-times to x. (Hence, $\lceil 0 \rceil$ is the identity.)

$$\lceil n \rceil = \lambda f \lambda x . f^{(n)}(x)$$

where

$$f^{(n)}(e) = \begin{cases} e & \text{if } n = 0 \\ f(f^{(n-1)}(e)) & \text{otherwise} \end{cases}$$

Remark

$\lceil n \rceil$ is a normal form!

Successor and Predecessor

- The successor function adds an application:

$$succ = \lambda n.\lambda f \lambda x.n\ f\ (f\ x)$$

Successor and Predecessor

- The successor function adds an application:

$$succ = \lambda n.\lambda f \lambda x.n\ f\ (f\ x)$$

- The predecessor is somewhat more complicated:

$$pred = \lambda x.\lambda y.\lambda z.x\ (\lambda p.\lambda q.q\ (p\ y))\ ((\lambda x.\lambda y.x)\ z)\ (\lambda x.x)$$

(A proof that it actually does subtract one from a Church numeral is a worthwhile exercise.)

Successor and Predecessor

- The successor function adds an application:

$$succ = \lambda n.\lambda f.\lambda x.n\ f\ (f\ x)$$

- The predecessor is somewhat more complicated:

$$pred = \lambda x.\lambda y.\lambda z.x\ (\lambda p.\lambda q.q\ (p\ y))\ ((\lambda x.\lambda y.x)\ z)\ (\lambda x.x)$$

  (A proof that it actually does subtract one from a Church numeral is a worthwhile exercise.)

- Testing for zero

$$zero? = \lambda n.n\ (\lambda x.false)\ true$$

Example Calculation

$$\begin{aligned}
\mathit{zero?}\ \lceil 0 \rceil &= (\lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true})\ \lceil 0 \rceil \\
&\rightarrow_\beta \lceil 0 \rceil\ (\lambda x.\mathit{false})\ \mathit{true} \\
&= (\lambda f.\lambda x.x)\ (\lambda x.\mathit{false})\ \mathit{true} \\
&\rightarrow_\beta (\lambda x.x)\ \mathit{true} \\
&\rightarrow_\beta \mathit{true}
\end{aligned}$$

Fixpoint Theorem

Every lambda term has a fixpoint.
That is, for every lambda term f there is a lambda term e with $f\ e \overset{*}{\leftrightarrow}_\beta e$.

Recursion

Fixpoint Theorem

Every lambda term has a fixpoint.
That is, for every lambda term f there is a lambda term e with $f\ e \overset{*}{\leftrightarrow}_\beta e$.

Proof:

Choose $e := Y\ f$ with

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)).$$

Then:

$$\begin{aligned}
Y\ F &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\ F \\
&\rightarrow_\beta (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\
&\rightarrow_\beta F\ ((\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))) \\
&\leftarrow_\beta F\ ((\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ F) \\
&= F\ (Y\ F)
\end{aligned}$$

Example Fixpoint Calculation

As an example, consider expressing the recursive definition of the factorial function

$$fac\ n = if\ (zero?\ n)\ \lceil 1 \rceil\ times\ n\ (fac\ (pred\ n))$$

where *times* and *pred* are multiplication and predecessor functions. An equivalent non-recursive definition can be found using the fixpoint combinator.

$$fac' = Y\ (\lambda f\ n.if\ (zero?\ n)\ \lceil 1 \rceil\ times\ n\ (f\ (pred\ n)))$$

Pairs

A pair can be encoded as a function that takes a projection function and applies it to the components of the pair. Hence, the selectors take a pair and apply it to the appropriate projection function.

$$
\begin{aligned}
\mathit{pair} &= \lambda xyt.t\ x\ y \\
\mathit{fst} &= \lambda p.p\ \lambda xy.x \\
\mathit{snd} &= \lambda p.p\ \lambda xy.y
\end{aligned}
$$

Pairs can be used to systematically derive a subtraction function that is "obviously" correct.

Plan

Weak Head-Normal Form

The Problem with Normal Forms

- difficult to compute efficiently
- full subsitution is complicated and expensive
- success depends on evaluation order

In practice, lambda terms are evaluated to the point where they are abstractions; it is not necessary to evaluate anything "inside the lambda."

Evaluation Strategies

Definition: Weak Head-Normal Form (WHNF)

- An abstraction is a **value** (or **weak head-normal form**).
- Any other term is a **non-value** (or **expression juxtaposition**).

Remark

A term need not have a WHNF: $(\lambda x\,.x\ x)\ (\lambda x\,.x\ x)$

Definition

An **evaluation strategy** is an algorithm to reduce a lambda term to its weak head-normal form (if one exists).

Contexts

- Evaluation strategy = algorithm that finds the next (beta) redex.
- Can be specified succinctly using **evaluation contexts**.
- Evaluation contexts are special contexts.

Definition: Context

A **context** is a lambda term with a **hole**.

$$C ::= [\,] \mid \lambda x\,.\,C \mid C\ e \mid e\ C$$

Hole Filling

Definition

Given a context C and a term f, the **hole filling** operation $C[f]$ is defined by

$$
\begin{aligned}
[\,][f] &= f \\
(\lambda x\,.\,C)[f] &= \lambda x\,.\,C[f] \\
(C\ e)[f] &= (C[f])\ e \\
(e\ C)[f] &= e\ (C[f])
\end{aligned}
$$

Hole Filling

Definition

Given a context C and a term f, the **hole filling** operation $C[f]$ is defined by

$$
\begin{aligned}
[\,][f] &= f \\
(\lambda x\,.\,C)[f] &= \lambda x\,.\,C[f] \\
(C\ e)[f] &= (C[f])\ e \\
(e\ C)[f] &= e\ (C[f])
\end{aligned}
$$

Examples

$$(\lambda x\,.\,[\,])[\lambda y\,.\,y] = \lambda x\,.\,\lambda y\,.\,y$$ like substitution

$$(\lambda x\,.\,[\,])[x] = \lambda x\,.\,x$$ unlike: variable in filling term may be captured

Evaluation Contexts

Given a reduction as a pair of redex (lhs) and contractum (rhs) (e.g., beta reduction)

$$(\lambda x \,.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]$$

define a grammar of **evaluation contexts** E and extend reduction by closing under contexts described by E:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

Different evaluation contexts describe different evaluation strategies.

Evaluation Contexts: Examples

Call-by-name Lambda Calculus

Reduction relation: full beta

$$(\lambda x \,.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]$$

Evaluation contexts

$$E_n ::= [\,] \mid E_n\ e$$

Evaluation Contexts: Examples

Call-by-name Lambda Calculus

Reduction relation: beta value

$$v ::= \lambda x \,.\, e \qquad \text{grammar of values}$$

$$(\lambda x \,.\, e)\; v \longrightarrow e[x \mapsto v] \qquad \text{argument must be value}$$

Evaluation contexts

$$E_v ::= [\,] \mid E_v\; e \mid v\; E_v$$

Deterministic Evaluation

Unique Decomposition

Suppose that E is a language of evaluation contexts.
If e is a term, then either

1. e is a value
2. $e \equiv E[r]$ for some unique evaluation context E and redex r
3. $e \equiv E[f]$ for some unique evaluation context E and irreducible term f

Deterministic Evaluation

Unique Decomposition

Suppose that E is a language of evaluation contexts.
If e is a term, then either

1. e is a value
2. $e \equiv E[r]$ for some unique evaluation context E and redex r
3. $e \equiv E[f]$ for some unique evaluation context E and irreducible term f

Remarks

- Would like to stay with (1) and (2).
- Restriction to closed terms removed case $E[x]$ from (3).
- Remaining cases in (3) can be avoided by **typing**.

Plan

Lambda Calculus with Constants

- Computing with Church numerals and the fixpoint combinator is unrealistic
- Real use efficient implementations of datatypes and recursion
- One way of modeling these implementations: add constants c!

Lambda Calculus with Constants

Syntax

Add infinitely many constants c to the syntax

$$e ::= \color{red}{c} \mid x \mid \lambda x\,.\,e \mid e\ e$$

Lambda Calculus with Constants

Syntax

Add infinitely many constants c to the syntax

$$e ::= c \mid x \mid \lambda x \,.\, e \mid e\ e$$

Reduction

Call-by-value = beta-value with evaluation contexts E_v

$$v ::= c \mid \lambda x \,.\, e \qquad \text{constants are values (WHNF)}$$

Behavior of constants defined by δ reductions

$$c\ v \longrightarrow_\delta \delta^c(v) \qquad \text{if } \delta^c \text{ defined}$$

where each $\delta^c : \mathsf{Val} \hookrightarrow \mathsf{Val}$ is a partial function on values.

Lambda Calculus with Constants (Example)

Applied Lambda Calculus with Integers and Addition

- Constants $\lceil n \rceil$ for each integer n (without reduction rules)
- A constant $+$ and constants $+_n$ for each integer

Reduction rules

$$\begin{aligned} \delta^{+} \lceil n \rceil &= +_n \\ \delta^{+_n} \lceil m \rceil &= \lceil n + m \rceil \end{aligned}$$

The set of values

$$v \quad ::= \quad \lceil n \rceil \mid + \mid +_n \mid \lambda x . e$$

A New Source of Errors

Stuck Terms

In an applied lambda calculus, there are usually terms which cannot be evaluated further although they are not in weak head-normal form. These terms are called **stuck terms**. They are regarded as execution errors because they amount to misinterpretation of data.

A New Source of Errors

Stuck Terms

In an applied lambda calculus, there are usually terms which cannot be evaluated further although they are not in weak head-normal form. These terms are called **stuck terms**. They are regarded as execution errors because they amount to misinterpretation of data.

Example

$\lceil 5 \rceil\ v$	number used as a function
$+\ (\lambda x.e)\ v$	operand out of domain
$\mathit{if}\ (\lambda x.e)\ \mathit{then}\ e_1\ \mathit{else}\ e_2$	type mismatch
$\mathit{if}\ \lceil 42 \rceil\ \mathit{then}\ e_1\ \mathit{else}\ e_2$	type mismatch

Avoiding Misinterpretation Errors by Typing

Dynamic Typing

- the compiler generates code that tests all operands before it executes an operation
- every value must be equipped with sufficient type information at run time

Static Typing

- impose a typing discipline that rules out programs that may lead to execution errors
- requires design and implementation of a type checker
- no run-time overhead

Strong Typing vs Weak Typing

Strong Typing

In a strongly typed language, each value has one designated type and only operations for this particular type apply to the value.

Weak Typing

Weakly typed languages have a notion of conversion (or *coercion*) that silently converts unsuitable operands into arguments suitable for an operation.

Combining Concepts

- A language can be strongly typed with a dynamic typing discipline (*e.g.*, Racket, Python).
- It can be weakly typed with a static typing discipline (old versions of the C language, PL/1).
- Popular combinations are either strong, static typing (Haskell, ML) or weak, dynamic typing (JavaScript).
- Java is special because a strong, static type discipline is meant to imply that no type mismatches can occur at runtime. However, this is not true in Java due to the presence (and wide use) of type casts in the language.