Introduction to the λ-calculus

Pedro Vasconcelos

February 16, 2024

What is the λ-calculus?

- A universal model of computation, i.e. equivalent to the Turing machine
- Unlike the TM, λ-calculus is also a model for programming languages:
  - variable scope
  - evaluation order
  - data structures
  - recursion
  - ...
- Functional languages can be seen computational implementations of the λ-calculus (Landin, 1964)

Bibliography

1. *Foundations of Functional Programming*, Lectures notes by Lawrence C. Paulson `https://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf`
2. *Lambda Calculi: a guide for computer scientists*, Chris Hankin, Graduate Texts in Computer Science, Oxford University Press

Plan

Plan

Terms of the λ-calculus

$x, y, z, \ldots$	a single variable is a term;
$(\lambda x\, M)$	is a term if x is a variable and M is a term;
$(M\, N)$	is a term if M e N are term.

examples of terms
y
$(\lambda x\, y)$
$((\lambda x\, y)\,(\lambda x\,(\lambda x\, y)))$
$(\lambda y\,(\lambda x\,(y\,(y\, x))))$

examples of non-terms
$()$
$x \lambda y$
$x(y)$
$(\lambda x\,(\lambda y\, y)$

Interpretation of the λ-calculus

$(\lambda x\, M)$ is the *abstraction* of x in M.

$(M\, N)$ is the *aplication* of M to the argument N.

Examples:

$(\lambda x\, x)$ is the *identity function*, i.e. the function for each x yields x

$(\lambda x\, (\lambda y\, x))$ is the function that for each x yeilds another function that for each y yield x

- No distiction between “data” and “programs”
- No constants (e.g. numbers)
- Everything is a λ-term!

Conventions regarding parenthesis

$$\begin{array}{rcl} \lambda x_1 x_2 \ldots x_n.\, M & \equiv & (\lambda x_1\, (\lambda x_2\, \ldots (\lambda x_n\, M) \ldots)) \\ (M_1\, M_2\, \ldots M_n) & \equiv & (\ldots (M_1\, M_2) \ldots M_n) \end{array}$$

Examples:

$$\begin{array}{rcl} \lambda xy.\, x & \equiv & (\lambda x\, (\lambda y\, x)) \\ \lambda xyz.\, xz(yz) & \equiv & (\lambda x\, (\lambda y\, (\lambda z\, ((x\, z)\, (y\, z))))) \end{array}$$

Free and bound ocorrences I

Variable x is in bound in $(\lambda x\, M)$.

An occurrence that is not bound is called free.

$$(\lambda z\,(\lambda x\,(y\; x))) \quad x \text{ bound},\ \ y \text{ free}$$

NB: a single variable may occur both free and bound in a single term.

$$((\lambda x\; x)\,(\lambda y\; x)) \quad x \text{ bound},\ \ x \text{ free}$$

Free and bound ocorrences II

$BV(M)$ set of bound variables in M

$FV(M)$ set of free variables in M

$$BV(\lambda z\,(\lambda x\,(y\,x))) = \{x,\, z\}$$
$$FV(\lambda z\,(\lambda x\,(y\,x))) = \{y\}$$
$$BV((\lambda x\,x)\,(\lambda y\,x)) = \{x,\, y\}$$
$$FV((\lambda x\,x)\,(\lambda y\,x)) = \{x\}$$

Defined by recursion over the term (see the bibliography).

Substitutions

$M[N/x]$ is the term resulting from *substituting* free occurrences of x in M by N.

$$(\lambda x\, y)[(z\, z)/y] \equiv (\lambda x\, (z\, z))$$
$$(\lambda x\, y)[(z\, z)/x] \equiv (\lambda x\, y)$$

Note that we substitute only free occurences of x.

Change of bound variables

The names of bound variables in programming languages are not significant, i.e. the following two functions are equivalent.

```
int f(int x,int y) {
  return x+y;
}
```

```
int f(int a,int b) {
  return a+b;
}
```

The notion of α-equivalence in the λ-calculus formalizes this equivalence.

α-Conversion

$$(\lambda x\, M) \to_\alpha (\lambda y\, M[y/x]) \qquad \text{if } y \notin BV(M) \cup FV(M)$$

Examples:

$$\lambda x.\, xy \to_\alpha \lambda z.\, xy[z/x] \equiv \lambda z.\, zy$$

$$\lambda x.\, xy \not\to_\alpha \lambda y.\, xy[y/x] \equiv \lambda y.\, yy \qquad \text{because } y \in FV(xy)$$

α-Equivalence

We will consider $M \simeq N$ if $M \to_\alpha N$ or $N \to_\alpha M$.

Example:

$$\lambda x.\, xy \simeq \lambda z.\, zy$$

More generally:

$$M \simeq N \quad \text{if} \quad M \to_\alpha^* N \text{ or } N \to_\alpha^*$$

where $\to_\alpha^*$ is the transitive closure of $\to_\alpha$.

Plan

1. Syntax
2. Reduction and normalization
3. Reduction strategies
4. Computation

β-Conversion

$$((\lambda x\, M)\, N) \rightarrow_\beta M[N/x] \qquad \text{if } BV(M) \cap FV(N) = \emptyset$$

Example:

$$((\lambda x\, \underbrace{(x\, x)}_{M})\, \underbrace{(y\, z)}_{N}) \rightarrow_\beta (x\, x)[(y\, z)/x] \equiv ((y\, z)\, (y\, z))$$

Corresponds to function call:

- x is the formal parameter;
- M is the function body;
- N is the argument.

Variable capture

The condition $BV(M) \cap FV(N) = \emptyset$ is required to avoid variable capture:

$$((\lambda x \overbrace{(\lambda y\, x)}^{M}) \overbrace{\ y\ }^{N}) \to_\beta (\lambda y\, x)[y/x] \qquad y \in BV(M) \cap FV(N) \neq \emptyset$$

$$\equiv (\lambda y\, y)$$

We may use α-conversions to avoid the capture of variable y:

$$((\lambda x\, (\lambda y\, x))\, y) \to_\alpha ((\lambda x \overbrace{(\lambda z\, x)}^{M}) \overbrace{\ y\ }^{N})$$

$$\to_\beta (\lambda z\, x)[x/y] \qquad BV(M) \cap FV(N) = \emptyset$$

$$\equiv (\lambda z\, y)$$

η-Conversion

$$(\lambda x\,(M\,x)) \to_\eta M$$

- Simplifies a redudant abstraction:

$$((\lambda x\,(M\,x))\,N) \to_\beta (M\,N) \quad \text{logo} \quad (\lambda x\,(M\,x)) \simeq M$$

- Only necessary to ensure the uniqueness of the normal form (further ahead)
- Not necessary for the implementation of programming languages

Currying

We don't need abstractions with 2 or more arguments:

$$\lambda xy.\, M \quad \equiv \quad (\lambda x\, (\lambda y\, M))$$

Arguments are substituted one at a time:

$$\begin{aligned} ((\lambda xy.\, M)\, P\, Q) &\equiv (((\lambda x\, (\lambda y\, M))\, P)\, Q) \\ &\to_\beta ((\lambda y\, M)[P/x]\, Q) \\ &\to_\beta M[P/x][Q/y] \end{aligned}$$

This encoding is called "*currying*" to honor the logician Haskell Curry (thought it has been introduced before by Schönfinkel e Frege).

Reductions

We write $M \to N$ if M reduces in a single β or η step to N.

We write $M \twoheadrightarrow N$ for the multi-step reductions ($\to^*$).

Equality

We write $M = N$ if M can be converted in N using zero or more *reductions* or *expansions*; i.e., the relation $(\to \cup \to^{-1})^*$.

Example:

$$a((\lambda y.\, by)c) = (\lambda x.\, ax)(bc)$$

because

$$a((\lambda y.\, by)c) \to a(bc) \leftarrow (\lambda x.\, ax)(bc)$$

Intuition: if $M = N$ then M and N are terms with identical "result".

Normal form

M is in normal form if there is no N such that $M \to N$.

M admits normal forms N if $M \twoheadrightarrow N$ and N is in normal form.

Example:

$$(\lambda x.\, a\, x)\,((\lambda y.\, by)\, c) \to a((\lambda y.\, by)\, c) \to a(bc) \not\to$$

Hence: $(\lambda x.\, a\, x)\,((\lambda y.\, by)\, c)$ admits normal form $a(bc)$.

Intuition: the result of the computation $(\lambda x.\, a\, x)\,((\lambda y.\, by)\, c)$ is $a(bc)$.

Terms with no normal form

Not all terms admit a normal form:

$$\begin{aligned}\Omega &\equiv ((\lambda x.\, x\, x)\,(\lambda x.\, x\, x)) \\ &\to_\beta (x\, x)[(\lambda x.\, x\, x)/x] \\ &\equiv ((\lambda x.\, x\, x)\,(\lambda x.\, x\, x)) \equiv \Omega\end{aligned}$$

Hence:

$$\Omega \to \Omega \to \Omega \to \cdots$$

i.e. Ω is a non-terminating computation.

Confluence

We can perform reductions in different orders.

Example:

$$\underline{(\lambda x.\, a\, x)\, ((\lambda y.\, b y)\, c)} \to a(\underline{(\lambda y.\, b y)\, c}) \to a(bc) \not\to$$

$$(\lambda x.\, a\, x)\, (\underline{(\lambda y.\, b y)\, c}) \to \underline{(\lambda x.\, a\, x)\, (bc)} \to a(bc) \not\to$$

Q: Do we always get the same normal form?

A: Yes: this is the Church-Rosser theorem

Confluence

(Church-Rosser)

If $M = N$ then there exists L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$.

The proof is based on the following

Diamond property

If
$$\begin{tikzcd}[column sep=small] & M \arrow[dl, two heads] \arrow[dr, two heads] & \\ M_1 & & M_2 \end{tikzcd}$$
then exists L such that
$$\begin{tikzcd}[column sep=small] M_1 \arrow[dr, two heads] & & M_2 \arrow[dl, two heads] \\ & L & \end{tikzcd}.$$

For more details: see the bibliography.

Plan

Reduction strategies

How should we reduce $(M\,N) \twoheadrightarrow P$?

normal order: reduce M and substitute N without reducing it

1. $M \twoheadrightarrow (\lambda x\, M')$
2. $M'[N/x] \twoheadrightarrow P$

applicative order: reduce both M and N before the substution

1. $M \twoheadrightarrow (\lambda x\, M')$
2. $N \twoheadrightarrow N'$
3. $M'[N'/x] \twoheadrightarrow P$

Facts about reductions

1. If reduction terminates with both strategies then they reach the same normal form
2. If a term admits normal form, this is obtained by reduction in normal order
3. Reduction in applicative order may fail to terminate even when there is a normal form
4. Reduction in normal order may duplicate computations, i.e. reduce the same sub-term multiple times

Applicative order: non-termination

Let $\Omega \equiv ((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))$; let use reduce

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

Normal order

$$\underline{((\lambda x.\, y)\, \Omega)} \rightarrow_\beta y \quad \text{normal form}$$

Applicative order

$$((\lambda x.\, y)\, \underline{\Omega}) \rightarrow_\beta ((\lambda x.\, y)\, \underline{\Omega}) \rightarrow_\beta \cdots \quad \text{non-terminating}$$

Normal order: duplicate computation

Supose we have a term **mult** such that

$$\textbf{mult}\ N\ M \to N \times M$$

for some terms N, M encoding the natural numbers (we will see these further ahead).

Define

$$\textbf{sqr} \equiv \lambda x.\,\textbf{mult}\,x\,x$$

Let us reduce

$$\textbf{sqr}\,(\textbf{sqr}\,N)$$

Normal order: duplicate computation

Aplicative order:

$$\textbf{sqr}\,(\textbf{sqr}\,N) \rightarrow \textbf{sqr}\,(\textbf{mult}\,N\,N) \rightarrow \textbf{sqr}\,N^2 \rightarrow \textbf{mul}\,N^2\,N^2$$

Normal order:

$$\begin{aligned}\textbf{sqr}\,(\textbf{sqr}\,N) &\rightarrow \textbf{mult}\,(\textbf{sqr}\,N)\,(\textbf{sqr}\,N)\\ &\rightarrow \textbf{mult}\,\underbrace{(\textbf{mult}\,N\,N)\,(\textbf{mult}\,N\,N)}_{\text{duplication}}\end{aligned}$$

Plan

Computation using the λ-calculus

The λ-calculus is a universal model for computation: any computable function (i.e. expressable using a Turing machine) can be encoded in λ-calculus.

Computation using the λ-calculus

The pure λ-calculus does not include booleans, integers, lists, etc. as primitives.

This omission is not limiting: such structures can be defined in the calculus itself.

However: implementations of functional languages typically use optimized representations for eficiency.

Booleans

Define:

$$\begin{aligned} \textbf{true} &\equiv \lambda xy.\, x \\ \textbf{false} &\equiv \lambda xy.\, y \\ \textbf{if} &\equiv \lambda pxy.\, pxy \end{aligned}$$

Then:

$$\begin{aligned} \textbf{if}\,\textbf{true}\, M\, N &\twoheadrightarrow M \\ \textbf{if}\,\textbf{false}\, M\, N &\twoheadrightarrow N \end{aligned}$$

Execise: check the reductions above.

Ordered pairs

One *constructor* and two *projections*:

$$\begin{aligned}\textbf{pair} &\equiv \lambda xyf.\, fxy\\ \textbf{fst} &\equiv \lambda p.\, p\,\textbf{true}\\ \textbf{snd} &\equiv \lambda p.\, p\,\textbf{false}\end{aligned}$$

Then:

$$\begin{aligned}\textbf{fst}\,(\textbf{pair}\,M\,N) &\twoheadrightarrow \textbf{fst}\,(\lambda f.\, f\,M\,N)\\ &\rightarrow (\lambda f.\, f\,M\,N)\,\textbf{true}\\ &\rightarrow \textbf{true}\,M\,N\\ &\twoheadrightarrow M\end{aligned}$$

Analogously: $\textbf{snd}\,(\textbf{pair}\,M\,N) \twoheadrightarrow N$.

Natural numbers

Using Church numerals:

$$\begin{aligned}
\underline{0} &\equiv \lambda f x.\, x \\
\underline{1} &\equiv \lambda f x.\, f\, x \\
\underline{2} &\equiv \lambda f x.\, f\, (f\, x) \\
&\vdots \\
\underline{n} &\equiv \lambda f x.\, \underbrace{f\, (\ldots (f}_{n \text{ times}}\ x) \ldots)
\end{aligned}$$

Intuition: $\underline{n}$ is a term that iterates a function n times.

Arithmetic operations

$$\begin{aligned}
\textbf{succ} &\equiv \lambda nfx.\, f\,(n\,f\,x) \\
\textbf{iszero} &\equiv \lambda n.\, n\,(\lambda x.\, \textbf{false})\,\textbf{true} \\
\textbf{add} &\equiv \lambda mnfx.\, m\,f\,(n\,f\,x)
\end{aligned}$$

Check:

$$\begin{aligned}
\textbf{succ}\,\underline{n} &\twoheadrightarrow \underline{n+1} \\
\textbf{iszero}\,\underline{0} &\twoheadrightarrow \textbf{true} \\
\textbf{iszero}\,(\underline{n+1}) &\twoheadrightarrow \textbf{false} \\
\textbf{add}\,\underline{n}\,\underline{m} &\twoheadrightarrow \underline{n+m}
\end{aligned}$$

Analogously: subtraction, multiplication, exponentiation, etc.

Lists

$$[x_1,\ x_2, \ldots,\ x_n] \quad \simeq \quad \mathbf{cons}\, x_1\, (\mathbf{cons}\, x_2\, (\ldots (\mathbf{cons}\, x_n\, \mathbf{nil}) \ldots))$$

Two constructors, check for the empty list and two projections.

$$\begin{aligned}
\mathbf{nil} &\equiv \lambda z.\, z \\
\mathbf{cons} &\equiv \lambda xy.\, \mathbf{pair}\, \mathbf{false}\, (\mathbf{pair}\, x\, y) \\
\mathbf{null} &\equiv \mathbf{fst} \\
\mathbf{hd} &\equiv \lambda z.\, \mathbf{fst}\, (\mathbf{snd}\, z) \\
\mathbf{tl} &\equiv \lambda z.\, \mathbf{snd}\, (\mathbf{snd}\, z)
\end{aligned}$$

Lists

Verify:

$$\textbf{null}\,\textbf{nil} \twoheadrightarrow \textbf{true} \tag{1}$$

$$\textbf{null}\,(\textbf{cons}\,M\,N) \twoheadrightarrow \textbf{false} \tag{2}$$

$$\textbf{hd}\,(\textbf{cons}\,M\,N) \twoheadrightarrow M \tag{3}$$

$$\textbf{tl}\,(\textbf{cons}\,M\,N) \twoheadrightarrow N \tag{4}$$

NB: (2), (3), (4) result from the properties of pairs, but (1) does not.

Declarations

$$\textbf{let } x = M \textbf{ in } N$$

Example:

$$\textbf{let } f = \lambda x.\, \textbf{add } x\ x \textbf{ in } \lambda x.\, f\ (f\ x)$$

Translation to the λ-calculus

$$\textbf{let } x = M \textbf{ in } N \;\equiv\; (\lambda x.\, N)\, M$$

Then:

$$\textbf{let } x = M \textbf{ in } N \;\twoheadrightarrow\; N[M/x]$$

Nested declarations

No extra syntax needed:

$$\begin{aligned} & \textbf{let } \{x = M;\ y = N\} \textbf{ in } P \\ \equiv\ & \textbf{let } x = M \textbf{ in } (\textbf{let } y = N \textbf{ in } P) \end{aligned}$$

Recursive declarations

First attempt:

$$\begin{array}{l}\textbf{let } f = \lambda x.\, \textbf{if } (\textbf{iszero}\, x)\ \underline{1}\ (\textbf{mult}\ x\ (f\ (\textbf{sub}\ x\ \underline{1})))\\ \ \textbf{in } f\ \underline{5}\end{array}$$

Translation:

$$(\lambda f.\, f\ \underline{5})\ (\lambda x.\, \textbf{if } (\textbf{iszero}\, x)\ \underline{1}\ (\textbf{mult}\ x\ ({\color{red}f}\ (\textbf{sub}\ x\ \underline{1}))))$$

This does **not** define a recursive function because *f* occurs free in the body of the definition.

Fixed-point combinators

Solution: use a fixed-point combinator i.e. a term $\mathbf{Y}$ such that

$$\mathbf{Y}\,F = F\,(\mathbf{Y}\,F) \quad \text{for any term } F$$

Then we can define recursive factorial as:

$$\begin{array}{l} \mathbf{let}\ f = \mathbf{Y}\,(\lambda g\,x.\,\mathbf{if}\ (\mathbf{iszero}\ x)\ \underline{1}\ (\mathbf{mult}\ x\ (g\ (\mathbf{sub}\ x\ \underline{1})))) \\ \ \mathbf{in}\ f\ \underline{5} \end{array}$$

Note that g occurs bound in the body of the function.

Fixed-point combinators

Assume:

$$\mathbf{Y}\ F = F\ (\mathbf{Y}\ F) \quad \text{para qualquer } F$$
$$\text{fact} \equiv \mathbf{Y}\,(\lambda g\,x.\,\mathbf{if}\ (\mathbf{iszero}\ x)\ \underline{1}\ (\mathbf{mult}\ x\ (g\ (\mathbf{sub}\ x\ \underline{1}))))$$

Let use compute:

$$\begin{aligned}
\text{fact}\ \underline{5} &\equiv \mathbf{Y}\ (\lambda g\,x.\ \ldots)\ \underline{5} \\
&= (\lambda g\,x.\ \ldots)\ \underbrace{(\mathbf{Y}\ (\lambda g\,x.\ \ldots))}_{\text{fact}}\ \underline{5} \\
&\twoheadrightarrow \mathbf{if}\ (\mathbf{iszero}\ \underline{5})\ \underline{1}\ (\mathbf{mult}\ \underline{5}\ (\text{fact}\ (\mathbf{sub}\ \underline{5}\ \underline{1}))) \\
&\twoheadrightarrow \mathbf{if}\ \mathbf{false}\ \underline{1}\ (\mathbf{mult}\ \underline{5}\ (\text{fact}\ \underline{4})) \\
&\twoheadrightarrow \mathbf{mult}\ \underline{5}\ (\text{fact}\ \underline{4}) \\
&\twoheadrightarrow \mathbf{mult}\ \underline{5}\ (\mathbf{mult}\ \underline{4}\ (\ldots\ (\mathbf{mult}\ \underline{1}\ \underline{1})\ldots)) \equiv \underline{120}
\end{aligned}$$

Fixed-point combinators

$\mathbf{Y}$ can be defined in the pure λ-calculus (Haskell B. Curry):

$$\mathbf{Y} \equiv \lambda f.\,(\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))$$

Check:

$$\begin{aligned}
\mathbf{Y}\;F &\to (\lambda x.\, F(xx))\,(\lambda x.\, F(xx)) \\
&\to F((\lambda x.\, F(xx))\,(\lambda x.\, F(xx))) \\
&\leftarrow F(\mathbf{Y}\,F)
\end{aligned}$$

Hence

$$\mathbf{Y}\;F = F(\mathbf{Y}\,F)$$

There are infinitely many other fixed-point combinators (see the bibliography).