Foundations of Functional Programming Languages

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2016-2

Administrative Information

Course web page

`http://www1.eafit.edu.co/asr/courses/foundations-of-functional-programming-languages/`

Exams, lectures, etc.

See course web page.

Some References

- Barendregt, H. P. [1984] [2004]. The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier.
- Barendregt, Henk [1992]. Lambda Calculi with Types. In: Handbook of Logic in Computer Science. Ed. by Abramsky, S., Gabbay, Dov M. and Maibaum, T. S. E. Vol. 2. Clarendon Press, pp. 117–309.
- Barendregt, Henk and Barendsen, Erik [2000]. Introduction to Lambda Calculus. Revisited edition.
- Hindley, J. Roger and Seldin, Jonathan P. [2008]. Lambda-Calculus and Combinators. An Introduction. Cambridge University Press.
- Mitchell, John C. [1996]. Foundations for Programming Languages. MIT Press.

Some History

- Lambda calculus [Church 1933, 1941].
- Simply typed lambda calculus [Church 1940].
- ISWIM (If you See What I Mean) [Landin 1966].
- PCF [Plotkin 1977].

Functional Programming

A description

'A **functional program** consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. **Reduction** consists of replacing a part P of E by another expression P' according to the given rewrite rules. In schematic notation

$$E[P] \to E[P'],$$

provided that $P \to P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called **normal form** E^* of the expression E consists of the output of the given functional program.' [Barendregt and Barendsen 2000, p. 6]

Functional Programming

Example

From [Barendregt and Barendsen 2000, p. 6].

```
head (sort (["dog", "rabbit"] ++ sort ["mouse", "cat"]))
→ head (sort (["dog", "rabbit"] ++ ["cat", "mouse"]))
→ head (sort ["dog", "rabbit", "cat", "mouse"])
→ head ["cat", "dog", "mouse", "rabbit"]
→ "cat"
```

Lambda Calculus

Introduction

Alonzo Church
(1903 – 1995)

- Invented by Church around 1930s
- Intended a foundational theory based on functions and recursion
- Computability model
- Basis of untyped functional programming languages

Inconsistencies

- λ-calculus and logic [Rosser 1984, § 2].
- λ-calculus and set theory [Paulson 2000, § 4.6].

Application and Abstraction

Application

Application of the function M to argument N is denoted by MX (juxtaposition).

Application and Abstraction

Application

Application of the function M to argument N is denoted by MX (juxtaposition).

Abstraction

'If M is any formula containing the variable x, then $\lambda x[M]$ is a symbol for the function whose values are those given by the formula.' [Church 1932, p. 352]

Currying

Definition

'Adopting a device due to Schönfinkel, we treat a function of two variables as a function of one variable whose values are functions of one variable, and a function of three or more variables similarly.' [Church 1932, p. 352]. Such device is called **currying** after Haskell Curry.

Currying

Definition

'Adopting a device due to Schönfinkel, we treat a function of two variables as a function of one variable whose values are functions of one variable, and a function of three or more variables similarly.' [Church 1932, p. 352]. Such device is called **currying** after Haskell Curry.

Let $f(x, y)$ be a function of two variables. We can define two functions

$$\begin{aligned} F_x &:= \lambda y.f(x, y), \\ F &:= \lambda x.F_x. \end{aligned}$$

Then

$$\begin{aligned} (Fx)y &= F_x y \\ &= f(x, y). \end{aligned}$$

Lambda Terms

Definition

Let $V = \{v_0, v_1, \dots\}$ be a set of variables. The set of λ-terms, denoted Λ, is inductively defined by

$$x \in V \Rightarrow x \in \Lambda \qquad \text{(variable)}$$

$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda \qquad \text{(application)}$$

$$x \in V, M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda \qquad \text{(abstraction)}$$

Conventions

- $x, y, z, ...$ denote arbitrary variables; $M, N, L, ...$ denote arbitrary λ-terms.

Conventions

- $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary λ-terms.
- Application associates to the left

  $MN_1 \cdots N_k$ means $(\cdots((MN_1)N_2)\cdots N_k)$

Conventions

- $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary λ-terms.
- Application associates to the left

  $MN_1 \cdots N_k$ means $(\cdots((MN_1)N_2)\cdots N_k)$
- Abstraction associates to the right

  $\lambda x_1 x_2 \cdots x_n.M$ means $(\lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))\cdots)))$

Conventions

- $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary λ-terms.
- Application associates to the left

  $MN_1 \cdots N_k$ means $(\cdots((MN_1)N_2)\cdots N_k)$
- Abstraction associates to the right

  $\lambda x_1 x_2 \cdots x_n.M$ means $(\lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))\cdots)))$
- Application has higher precedence

  $\lambda x.PQ$ means $(\lambda x.(PQ))$

Alpha Congruence

$M \equiv N$ denotes that

i) M and N are the same term or

ii) M and N can be obtained from each other by renaming bound variables.

Alpha Congruence

$M \equiv N$ denotes that

i) M and N are the same term or

ii) M and N can be obtained from each other by renaming bound variables.

Examples

$$x \equiv x, \qquad \lambda x.xy \equiv \lambda z.zy, \qquad (\lambda x.x)z \equiv (\lambda y.y)x,$$
$$x \not\equiv y, \qquad \lambda x.xy \not\equiv \lambda y.yy, \qquad (\lambda x.x)z \not\equiv (\lambda x.y)z.$$

Substitution

Ottmann variable convention

'If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.' [Barendregt 2004, pp. 26 and E1]

Example

We write $y(\lambda xy'.xy'z)$ for $y(\lambda xy.xyz)$.

Substitution

Definition

The result of substituting N for the free occurrences of x in M, denoted $M[x := N]$, is recursively defined by:

$$
\begin{aligned}
x[x := N] &\equiv N, \\
y[x := N] &\equiv y, \text{ if } x \not\equiv y, \\
(M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_2[x := N]), \\
(\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]).
\end{aligned}
$$

Substitution

Definition

The result of substituting N for the free occurrences of x in M, denoted $M[x := N]$, is recursively defined by:

$$\begin{aligned} x[x := N] &\equiv N, \\ y[x := N] &\equiv y, \text{ if } x \not\equiv y, \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_2[x := N]), \\ (\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]). \end{aligned}$$

Remark

In the fourth clause it is not needed to say 'provided that $y \not\equiv x$ and $y \notin FV(N)$'. By the Ottmann variable convention this is the case.

The Lambda Calculus as a Formal Theory

A equational theory

- Formulae: $M = N$, where $M, N \in \Lambda$
- Axiom scheme

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

- Rules

  Equality is a congruence relation on Λ, that is,
  - Equality is a equivalence relation
  - Equality is compatible with the constructors of λ-terms

The Lambda Calculus as a Formal Theory

Example

Prove that $\lambda \vdash (\lambda xy.x)MN = M$ for all $M, N \in \Lambda$.*

1	$(\lambda xy.x)M = \lambda y.x[x := M] \equiv \lambda y.M$	(β)
2	$(\lambda xy.x)MN = (\lambda y.M)N$	(comp. 1)
3	$(\lambda y.M)N = M[y := N] \equiv M$	(β)
4	$(\lambda xy.x)MN = M$	(trans. 2,3)

*Note that by the Ottmann variable convention it means that $x, y \notin FV(M) \cup FV(N)$.

The Lambda Calculus as a Formal Theory

Theorem (Barendregt and Barendsen [2000], Lemma 2.9)

$\lambda \vdash (\lambda x_1 \cdots x_n.M)X_1 \cdots X_n = M[x_1 := X_1] \cdots [x_n := X_n]$.

Combinators

Examples

identity	$\mathbf{I} \equiv \lambda x.x$	$\mathbf{I}M = M$
fst	$\mathbf{K} \equiv \lambda xy.x$	$\mathbf{K}MN = M$
snd	$\mathbf{K}_* \equiv \lambda xy.y$	$\mathbf{K}_*MN = N$
stronger composition	$\mathbf{S} \equiv \lambda fgx.fx(gx)$	$\mathbf{S}LMN = LN(MN)$
composition	$\mathbf{B} \equiv \lambda fgx.f(gx)$	$\mathbf{B}LMN = L(MN)$
doubling	$\mathbf{W} \equiv \lambda fx.fxx$	$\mathbf{W}MN = MNN$

Reductions

Definitions

i) A binary relation R on Λ is called compatible if

$$M\ R\ N \Rightarrow (ZM)\ R\ (ZN),$$
$$(MZ)\ R\ (NZ) \text{ and}$$
$$(\lambda x.M)\ R\ (\lambda x.N).$$

Reductions

Definitions

i) A binary relation R on Λ is called compatible if

$$M\ R\ N \Rightarrow (ZM)\ R\ (ZN),$$
$$(MZ)\ R\ (NZ) \text{ and}$$
$$(\lambda x.M)\ R\ (\lambda x.N).$$

ii) A reduction relation on Λ is a compatible, reflexive and transitive relation.

Reductions

Definitions

i) A binary relation R on Λ is called compatible if

$$M\ R\ N \Rightarrow (ZM)\ R\ (ZN),$$
$$(MZ)\ R\ (NZ) \text{ and}$$
$$(\lambda x.M)\ R\ (\lambda x.N).$$

ii) A reduction relation on Λ is a compatible, reflexive and transitive relation.

iii) A congruence relation on Λ is a compatible equivalence relation.

Reductions

Definitions

Inductive definitions of the binary relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ:

i) 1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$
   2. $M \rightarrow_\beta N \Rightarrow ZM \rightarrow_\beta ZN$, $MZ \rightarrow_\beta NZ$ and $\lambda x.M \rightarrow_\beta \lambda x.N$

Reductions

Definitions

Inductive definitions of the binary relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ:

i)
1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$
2. $M \rightarrow_\beta N \Rightarrow ZM \rightarrow_\beta ZN$, $MZ \rightarrow_\beta NZ$ and $\lambda x.M \rightarrow_\beta \lambda x.N$

ii)
1. $M \twoheadrightarrow_\beta M$
2. $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$
3. $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$

Reductions

Definitions

Inductive definitions of the binary relations $\to_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ:

i)
1. $(\lambda x.M)N \to_\beta M[x := N]$
2. $M \to_\beta N \Rightarrow ZM \to_\beta ZN$, $MZ \to_\beta NZ$ and $\lambda x.M \to_\beta \lambda x.N$

ii)
1. $M \twoheadrightarrow_\beta M$
2. $M \to_\beta N \Rightarrow M \twoheadrightarrow_\beta N$
3. $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$

iii)
1. $M \twoheadrightarrow_\beta N \Rightarrow M =_\beta N$
2. $M =_\beta N \Rightarrow N =_\beta M$
3. $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$

Reductions

Informally, $M =_\beta N$ if M is connected to N via (without direction) $\to \beta$-arrows.*

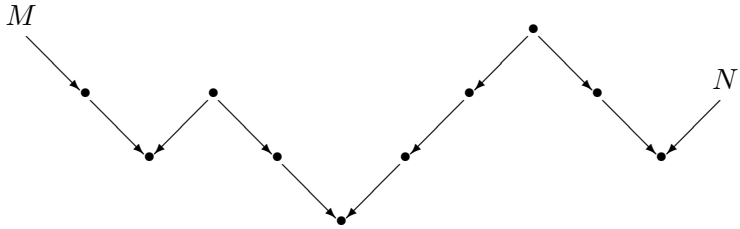

*Figure from [Barendregt and Barendsen 2000, p. 24].

Reductions

Theorem (Barendregt and Barendsen [2000], Proposition 4.5, p. 24)

$$M =_\beta N \Leftrightarrow \lambda \vdash M = N$$

Reductions

Theorem

If $M =_\beta N$, then there is an L such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$.*

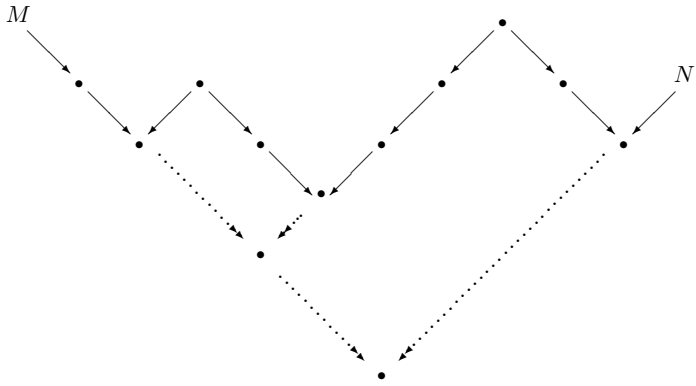

*Figure from [Barendregt and Barendsen 2000, p. 25].

Reductions

Theorem (Normalisation Theorem)

If M has a normal form, then iterated contraction of the leftmost redex leads to that normal form [Barendregt and Barendsen 2000, Theorem 4.22].

Reduction (Evaluation) Strategies

Call-by value (or eager evaluation)

The arguments are evaluated before substituting them into the body a the function.

Reduction (Evaluation) Strategies

Call-by value (or eager evaluation)

The arguments are evaluated before substituting them into the body a the function.

Call-by-name

The arguments are not evaluated before the function is called but only when needed. If an argument is used several times, it is re-evaluated each time it is needed.

Reduction (Evaluation) Strategies

Call-by value (or eager evaluation)

The arguments are evaluated before substituting them into the body a the function.

Call-by-name

The arguments are not evaluated before the function is called but only when needed. If an argument is used several times, it is re-evaluated each time it is needed.

Call-by-need (or lazy evaluation)

Memoized version of call-by-name.

Typed Lambda Calculus

Types

Example

$$\mathbb{V} ::= v \mid \mathbb{V}' \qquad \text{(type variables)}$$

$$\begin{aligned}
\mathbb{T} ::= {} & \mathbb{V} \\
\mid {} & \mathbb{C} && \text{(type constants)} \\
\mid {} & N_0 && \text{(empty type)} \\
\mid {} & N_1 && \text{(unit type)} \\
\mid {} & \mathbb{T} \to \mathbb{T} && \text{(function types)} \\
\mid {} & \mathbb{T} \times \mathbb{T} && \text{(product types)} \\
\mid {} & \mathbb{T} + \mathbb{T} && \text{(disjoint union types)}
\end{aligned}$$

What is a Type?

- Types as ranges of significance of propositional functions. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell 1938, Appendix B: The Doctrine of Types].

  In modern terminology, Rusell's types are domains of propositional functions.

  Example

  Let $\varphi(x)$ be the propositional function 'x is a prime number'. Then $\varphi(x)$ is a proposition only when its argument is a natural number.

$$\varphi : \mathbb{N} \to \{\mathrm{False}, \mathrm{True}\}$$
$$\varphi(x) = x \text{ is a prime number.}$$

What is a Type?

- Hoare's 'Notes on Data Structuring' [Hoare 1972, pp. 92-93]

What is a Type?

- Hoare's 'Notes on Data Structuring' [Hoare 1972, pp. 92-93]

  "*Thus there is a high degree of commonality in the use of the concept of type by mathematicians, logicians and programmers. The salient characteristics of the concept of type may be summarised:*"

  1. "*A type determines the class of values which may be assumed by a variable or expression.*"
  2. "*Every value belongs to one and only one type.*"
  3. "*The type of a value denoted by any constant, variable, or expression may be deduced from its form or context, without any knowledge of its value as computed at run time.*"

What is a Type?

- Hoare's 'Notes on Data Structuring' (cont.)

  4. "*Each operator expects operands of some fixed type, and delivers a result of some fixed type (usually the same). Where the same symbol is applied to several different types (e.g. + for addition of integers as well as reals), this symbol may be regarded as ambiguous, denoting several different actual operators. The resolution of such systematic ambiguity can always be made at compile time.*"
  5. "*The properties of the values of a type and of the primitive operations defined over them are specified by means of a set of axioms.*"

What is a Type?

- Hoare's 'Notes on Data Structuring' (cont.)

  6. "*Type information is used in a high-level language both to prevent or detect meaningless constructions in a program, and to determine the method of representing and manipulating data on a computer.*"

  7. "*The types in which we are interested are those already familiar to mathematicians; namely, Cartesian Products, Discriminated Unions, Sets, Functions, Sequences, and Recursive Structures.*"

What is a Type?

- The propositions-as-types principle (Curry-Howard correspondence)

What is a Type?

- The propositions-as-types principle (Curry-Howard correspondence)

  Three correspondence's levels [Wadler 2015]:

  1. Propositions as types

     "*for each proposition in the logic there is a corresponding type in the programming language—and vice versa*"

What is a Type?

- The propositions-as-types principle (Curry-Howard correspondence)

  Three correspondence's levels [Wadler 2015]:

  1. Propositions as types

     "*for each proposition in the logic there is a corresponding type in the programming language—and vice versa*"

  2. Proofs as programs

     "*for each proof of a given proposition, there is a program of the corresponding type—and vice versa*"

What is a Type?

- The propositions-as-types principle (Curry-Howard correspondence)

  Three correspondence's levels [Wadler 2015]:

  1. Propositions as types

     "*for each proposition in the logic there is a corresponding type in the programming language—and vice versa*"

  2. Proofs as programs

     "*for each proof of a given proposition, there is a program of the corresponding type—and vice versa*"

  3. Simplification of proofs as evaluation of programs

     "*for each way to simplify a proof there is a corresponding way to evaluate a program—and vice versa*"

What is a Type?

- "*A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.*"
  [Kiselyov and Shan 2008, p. 8]
- Homotopy Type Theory (HTT)

  Propositions are types, but not all types are propositions (e.g. higher-order inductive types)

Encoding Data in the Lambda Calculus*

*[Paulson 2000, Ch. 3].

Constructors, Predicates and Selectors

Definitions ([Landin 1964, p. 310])

*CO: structured definitions or constructed objects

Constructors, Predicates and Selectors

Definitions ([Landin 1964, p. 310])

i) Constructors: 'for constructing a CO* of given format from given components'.

ii) Predicates: 'for testing which of the various alternative formats (if there are alternatives) is possessed by a given CO'.

iii) Selectors: 'for selecting the various components of a given CO once its format is known'.

*CO: structured definitions or constructed objects

Constructors, Predicates and Selectors

Definitions ([Hoare 1972, p. 94])

Constructors, Predicates and Selectors

Definitions ([Hoare 1972, p. 94])

i) Constructors: 'permit the value of a structured type to be defined in terms of the values of the constituent types from which it is built.'

ii) Selectors: 'permit access to the component values of a structured type.'

Constructors, Predicates and Selectors

Definitions ([Hoare 1972, p. 94])

i) Constructors: 'permit the value of a structured type to be defined in terms of the values of the constituent types from which it is built.'

ii) Selectors: 'permit access to the component values of a structured type.'

Example

Constructors, predicates and selectors for the list data type. Whiteboard.

Booleans

Constructors

$$\mathbf{true} \equiv \mathbf{K} \equiv \lambda xy.x$$
$$\mathbf{false} \equiv \mathbf{K}_* \equiv \lambda xy.y$$

Selector

$$\mathbf{if} \equiv \lambda pxy.pxy$$

Conversion rules

$$\mathbf{if\ true}\ MN =_\beta M$$
$$\mathbf{if\ false}\ MN =_\beta N$$

Booleans

Constructors

$$\mathbf{true} \equiv \mathbf{K} \equiv \lambda xy.x$$
$$\mathbf{false} \equiv \mathbf{K}_* \equiv \lambda xy.y$$

Selector

$$\mathbf{if} \equiv \lambda pxy.pxy$$

Conversion rules

$$\mathbf{if\ true}\ MN =_\beta M$$
$$\mathbf{if\ false}\ MN =_\beta N$$

Remark

The above equations hold for all terms M and N (independently whether or not they have β-normal forms).

Booleans

Some operations

$$\mathbf{and} \equiv \lambda pq.\mathbf{if}\ p\ q\ \mathbf{false}$$
$$\mathbf{or} \equiv \lambda pq.\mathbf{if}\ p\ \mathbf{true}\ q$$
$$\mathbf{not} \equiv \lambda p.\mathbf{if}\ p\ \mathbf{false}\ \mathbf{true}$$

Ordered Pairs

Constructor

$$\mathbf{pair} \equiv \lambda xyf.fxy$$

Selectors

$$\mathbf{fst} \equiv \lambda p.p\ \mathbf{true}$$
$$\mathbf{snd} \equiv \lambda p.p\ \mathbf{false}$$

Conversion rules

$$\mathbf{fst}\ (\mathbf{pair}\ MN) =_\beta M$$
$$\mathbf{snd}\ (\mathbf{pair}\ MN) =_\beta N$$

Ordered Pairs

Constructor

$$\mathbf{pair} \equiv \lambda xyf.fxy$$

Selectors

$$\mathbf{fst} \equiv \lambda p.p\ \mathbf{true}$$
$$\mathbf{snd} \equiv \lambda p.p\ \mathbf{false}$$

Conversion rules

$$\mathbf{fst}\ (\mathbf{pair}\ MN) =_\beta M$$
$$\mathbf{snd}\ (\mathbf{pair}\ MN) =_\beta N$$

Remark

We can project any of the two components of $\mathbf{pair}\ MN$ (even if the other has no β-normal form).

Natural Numbers

Notation

$$
\begin{aligned}
X^0 Y &\equiv Y, \\
X^n Y &\equiv \underbrace{X(X(\cdots(X}_{n \text{ `}X\text{'s}} Y)\cdots)) \quad \text{if } n \geq 1.
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
\overline{0} &\equiv \lambda f x.x \\
\overline{1} &\equiv \lambda f x.fx \\
\overline{2} &\equiv \lambda f x.f(fx) \\
&\vdots \\
\overline{n} &\equiv \lambda f x.f^n x
\end{aligned}
$$

Natural Numbers

Constructors

$$\mathbf{zero} \equiv \overline{0} \equiv \lambda f x.x$$
$$\mathbf{succ} \equiv \lambda n f x.f(nfx)$$

Predicate

$$\mathbf{isZero} \equiv \lambda n.n(\lambda x.\mathbf{false})\ \mathbf{true}$$

Conversion rules

$$\mathbf{succ}\ \overline{n} =_\beta \overline{n+1}$$

$$\mathbf{isZero}\ \overline{0} =_\beta \mathbf{true}$$
$$\mathbf{isZero}\ \overline{n+1} =_\beta \mathbf{false}$$

Natural Numbers

Some operations

$$\mathbf{add} \equiv \lambda mnfx.mf(nfx)$$
$$\mathbf{mult} \equiv \lambda mnfx.m(nf)x$$

Conversion rules

$$\mathbf{add}\ \overline{m}\ \overline{n} =_\beta \overline{m+n}$$
$$\mathbf{mult}\ \overline{m}\ \overline{n} =_\beta \overline{m \times n}$$

Lists

Constructors

$$\mathbf{nil} \equiv \lambda z.z$$
$$\mathbf{cons} \equiv \lambda xy.\mathbf{pair}\ \mathbf{false}\ (\mathbf{pair}\ xy)$$

Selectors

$$\mathbf{head} \equiv \lambda z.\mathbf{fst}\ (\mathbf{snd}\ z)$$
$$\mathbf{tail} \equiv \lambda z.\mathbf{snd}\ (\mathbf{snd}\ z)$$

Conversion rules

$$\mathbf{head}\ (\mathbf{cons}\ MN) =_\beta M$$
$$\mathbf{tail}\ (\mathbf{cons}\ MN) =_\beta N$$

Lists

Predicate

$$\mathbf{isNull} \equiv \mathbf{fst}$$

Conversion rules

$$\mathbf{isNull}\ \mathbf{nil} =_\beta \mathbf{true}$$
$$\mathbf{isNull}\ (\mathbf{cons}\ MN) =_\beta \mathbf{false}$$

Writing Recursive Functions

Example

Recursive functions:

$$\begin{aligned}
\mathbf{fact}\ N &\equiv \mathbf{if}\ (\mathbf{isZero}\ N)\ \overline{1}\ (\mathbf{mult}\ N(\mathbf{fact}\ (\mathbf{pred}\ N))) \\
\mathbf{append}\ ZW &\equiv \mathbf{if}\ (\mathbf{isNull}\ Z)\ W\ (\mathbf{cons}\ (\mathbf{head}\ Z)(\mathbf{append}\ (\mathbf{tail}\ Z)W)) \\
\mathbf{zeros} &\equiv \mathbf{cons}\ \overline{0}\ \mathbf{zeros}
\end{aligned}$$

Writing the above function using the $\mathbf{Y}$ combinator:

$$\begin{aligned}
\mathbf{fact} &\equiv \mathbf{Y}\ (\lambda fn.\mathbf{if}\ (\mathbf{isZero}\ n)\ \overline{1}\ (\mathbf{mult}\ n(f(\mathbf{pred}\ N)))) \\
\mathbf{append} &\equiv \mathbf{Y}\ (\lambda fzw.\mathbf{if}\ (\mathbf{isNull}\ z)\ w\ (\mathbf{cons}\ (\mathbf{head}\ z)(f(\mathbf{tail}\ z)w))) \\
\mathbf{zeros} &\equiv \mathbf{Y}\ (\lambda f.\mathbf{cons}\ \overline{0}\ f)
\end{aligned}$$

ISWIM*

*[Paulson 2000, Ch. 6].

ISWIM: Lambda Calculus as a Programming Language

- ISWIM: If you See What I Mean
- Landin, P. J. [1966]. The Next 700 Programming Languages. Communications of the ACM 9.3, pp. 157–166. DOI: 10.1145/365230.365257

Local Declarations

Simple declaration

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

Local Declarations

Simple declaration

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

Example

- $\textbf{let } n = \overline{0} \textbf{ in succ } n$
- $\textbf{let } m = \overline{0} \textbf{ in } (\textbf{let } n = \overline{1} \textbf{ in add } m\ n)$

Local Declarations

Function declaration

$$\textbf{let } f x_1 \cdots x_k = M \textbf{ in } N \equiv (\lambda f.N)(\lambda x_1 \cdots x_k.M)$$

Local Declarations

Function declaration

$$\textbf{let } f x_1 \cdots x_k = M \textbf{ in } N \equiv (\lambda f.N)(\lambda x_1 \cdots x_k.M)$$

Example

$\textbf{let succ } n = \lambda f x.f(nfx) \textbf{ in succ } \overline{0}$

Local Declarations

Recursive declaration

$$\textbf{letrec } f x_1 \cdots x_k = M \textbf{ in } N \equiv (\lambda f.N)(\mathbf{Y}(\lambda f x_1 \cdots x_k.M))$$

Local Declarations

Recursive declaration

$$\textbf{letrec } f x_1 \cdots x_k = M \textbf{ in } N \equiv (\lambda f.N)(\mathbf{Y}(\lambda f x_1 \cdots x_k.M))$$

Example

$\textbf{letrec fact } n = \textbf{if } (\textbf{isZero } N)\ \overline{1}\ (\textbf{mult } N(\textbf{fact } (\textbf{pred } N)))\ \textbf{in fact } \overline{5}$

References

References

Barendregt, H. P. [1984] (2004). The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier (cit. on pp. 3, 21).

Barendregt, Henk (1992). Lambda Calculi with Types. In: Handbook of Logic in Computer Science. Ed. by Abramsky, S., Gabbay, Dov M. and Maibaum, T. S. E. Vol. 2. Clarendon Press, pp. 117–309 (cit. on p. 3).

Barendregt, Henk and Barendsen, Erik (2000). Introduction to Lambda Calculus. Revisited edition (cit. on pp. 3, 5, 6, 26, 34–37).

Church, Alonzo (1932). A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33.2, pp. 346–366. DOI: 10.2307/1968337 (cit. on pp. 10–13).

— (1933). A Set of Postulates for the Foundation of Logic (Second Paper). Annals of Mathematics 34.4, pp. 839–864. DOI: 10.2307/1968702 (cit. on p. 4).

— (1940). A Formulation of the Simple Theory of Types. The Journal of Symbolic Logic 5.2, pp. 55–68. DOI: 10.2307/2266170 (cit. on p. 4).

— (1941). The Calculi of Lambda-Conversion. Princenton University Press (cit. on p. 4).

References


Hindley, J. Roger and Seldin, Jonathan P. (2008). Lambda-Calculus and Combinators. An Introduction. Cambridge University Press (cit. on p. 3).


Hoare, C. A. R. (1972). Notes on Data Structuring. In: Structured Programming. Ed. by Dahl, O.-J., Disjkstra, E. W. and Hoare, C. A. R. Academic Press, pp. 83–174 (cit. on pp. 44, 45, 56–58).


Kiselyov, Oleg and Shan, Chung-chieh (2008). Interpreting Types as Abstract Values. Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008) (cit. on p. 52).


Landin, P. J. (1964). The Mechanical Evaluation of Expressions. The Computer Journal 6, pp. 308–320. DOI: 10.1093/comjnl/6.4.308 (cit. on pp. 54, 55).


— (1966). The Next 700 Programming Languages. Communications of the ACM 9.3, pp. 157–166. DOI: 10.1145/365230.365257 (cit. on pp. 4, 71).


Mitchell, John C. (1996). Foundations for Programming Languages. MIT Press (cit. on p. 3).


Paulson, Lawrence C. (2000). Foundations of Functional Programming. Lecture notes. URL: http://www.cl.cam.ac.uk/~lp15/ (visited on 10/06/2020) (cit. on pp. 9, 53, 70).

References


Plotkin, G. D. (1977). LCF Considered as a Programming Language. Theoretical Computer Science 5.3, pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5 (cit. on p. 4).


Rosser, J. Barkley (1984). Highlights of the History of Lambda-Calculus. Annals of the History of Computing 6.4, pp. 337–349. DOI: 10.1109/MAHC.1984.10040 (cit. on p. 9).


Russell, Bertrand [1903] (1938). The Principles of Mathematics. 2nd ed. W. W. Norton & Company, Inc (cit. on p. 43).


Wadler, Philip (2015). Propositions as Types. Communications of the ACM 58.12, pp. 75–84. DOI: 10.1145/2699407 (cit. on pp. 48–51).