λ-calculus

Matej Košík

12th November 2005

Disclaimer

This material is derived from this [1] book without notifying the publisher. Publishing it in any form would mean that you violated the existing copyright laws.

Parts of this material were also copied from [2], so they are covered by Creative Commons License.

The material you are viewing was created only for internal needs of our community.

Outline

Contents

1 Notions and properties of the λ-calculus **4**

1.1 Evaluation order independence 4

1.2 Referential transparency 5

1.3 Bound and free identifiers 6

2 λ-calculus programming language basics **7**

2.1 Syntax 7

2.2 Normal form 11

2.3 Multiple Parameters 12

2.4 Abbreviations 13

3 Basic data-types **14**

3.1 Booleans 14

3.2 Direct Product Types 16

3.3 Direct Sum of Types 19

3.4 Sequences 20

4 Auxiliary Declarations **22**

5 Recursive Declarations **23**

6 Natural Numbers **29**

The Context

Words like *algebra* and *algorithm* have their ancient origins in the celebrated book كِتَاب الجَبر وَالمُقَبِلَه written by اَبُو عَبد اللَه مُحَمَّد ابن مُوسَى ال خوَارِزمِي [3]

Here are some of the notions which participate on full formalization of the notion *computation*.

Algorithms are essential to the way computers process information, because a computer program is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task [2]. Thus, an algorithm can be considered to be any sequence of operations which can be performed by a Turing-complete system.

In computability theory a programming language or any other logical system is called *Turing-complete* if it has a computational power equivalent to a universal Turing machine. In other words, the system and the universal Turing machine can emulate each other [2].

Turing machine consists of [2]:

1. A tape which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special blank symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily expandable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the

blank symbol.

2. A head that can read and write symbols on the tape and move left and right.
3. A state register that stores the state of the Turing machine. The number of different states is always finite and there is one special start state with which the state register is initialized.
4. An action table (or transition function) that tells the machine what symbol to write, how to move the head ('L' for one step left, and 'R' for one step right) and what its new state will be, given the symbol it has just read on the tape and the state it is currently in. If there is no entry in the table for the current combination of symbol and state then the machine will halt.

Note that every part of the machine is finite; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

In computability theory a countable *set* is called *recursive* [2], computable or decidable if we can construct an algorithm which terminates after a finite amount of time and decides whether or not a given element belongs to the set.

In computability theory a countable *set* S is called *recursively enumerable* [2], computably enumerable, semi-decidable or provable if

- There is an algorithm that, when given an input — typically an integer or a tuple of integers or a sequence of characters — **eventually halts** if it is a member of S. Otherwise, there is no guarantee that the algorithm will halt.

Or, equivalently,

- There is an algorithm that "generates" the members of S. That means that its output is simply a list of the members of S: s1, s2, s3,... If necessary it runs forever.

Earlier, there was a tendency to define semantics of a programming language formally. One of the way how they accomplished it was by defining a function which projects any program in that language to a λ-expression (e.g. Actors and Algol). This has sense because semantics of the λ-calculus is clear (because it is such a simple language) and actually the term what is an algorithm was defined by means of λ-calculus or equivalent models (Turing machines, Post productions, Markov algorithms).

So, with the help of λ-calculus we can define what is an algorithm. However, not everything can be regarded as algorithm. Systems which operate continuously (they are not expected to halt) or systems interacting with other systems in their environment cannot be squashed to the notion of algorithm. The need for constructing such a system is becoming more and more common these days. They are not algorithms. Description of these systems—continuously operating and interacting with other systems in their environment—are best described by other formalisms [4, 5, 6]. But, these formalisms are not the topic of this presentation.

The Context

- computability theory
  - algebra and algorithms
  - **Turing completeness**
  - Turing machine
  - recursive set, recursively enumerable set
  - recursive function
  - primitively recursive function
- λ-calculus as a model for sequential computation
- Other models of sequential computation
  - Turing machines
  - Post productions
  - Markov algorithms
- λ-calculus and **operational semantics**
- continuously operating, interacting, concurrent systems

1 Notions and properties of the λ-calculus

Notions and properties of the λ-calculus

- evaluation
- evaluation order independence [1, §1.3]
- referential transparency [1, §1.4]

1.1 Evaluation order independence

To evaluate some expression means to extract its value. Evaluation of the expression $6 \times 2 + 2$ gives us the number 14. Could we evaluate also expression $(2ax + b)(2ax + c)$? No, unless we know the values of a, b, c, and x, i.e. *the context* of this expression. Assuming that $a = 3$, $b = 2$, $c = -1$ and $x = 2$ we are able to evaluate this expression to integer 154. The notion *evaluation order independence* means that it is possible to choose any order for reduction of sub-expressions provided that we do obey explicit and implicit parentheses.

Evaluation order independence

- We are free to choose any order in which we reduce expressions
- Of course, we have to obey:
  - explicit parentheses
  - implicit parentheses (operator priorities according to the contracts which were made in order to reduce the total number of parentheses)

Evaluation of $6 \times 2 + 2$

- $6 \times 2 + 2 \Rightarrow 12 + 2 \Rightarrow 14$

Left-to-right evaluation of $(2ax + b)(2ax + c)$

$(2ax + b)(2ax + c)$provided that $a \Rightarrow 3$, $b \Rightarrow 2$, $c \Rightarrow -1$ and $x \Rightarrow 2 \Rightarrow (2 \times 3 \times x + b)(2ax + c)$ $\Rightarrow (6 \times x + b)(2ax + c) \Rightarrow (6 \times 2 + b)(2ax + c) \Rightarrow (12 + b)(2ax + c) \Rightarrow (12 + 2)(2ax + c) \Rightarrow 14 \times (2ax + c)$ $\Rightarrow 14 \times (2 \times 3 \times x + c) \Rightarrow 14 \times (6 \times x + c) \Rightarrow 14 \times (6 \times 2 + c) \Rightarrow 14 \times (12 + c) \Rightarrow 14 \times (12 + (-1))$ $\Rightarrow 14 \times 11 \Rightarrow 154$

Right-to-left evaluation of $(2ax + b)(2ax + c)$

$(2ax + b)(2ax + c) \Rightarrow 154$provided that $a \Rightarrow 3$, $b \Rightarrow 2$, $c \Rightarrow -1$ and $x \Rightarrow 2 \Rightarrow (2ax + b)(2ax + (-1))$ $\Rightarrow (2ax + b)(2a \times 2 + (-1)) \Rightarrow (2ax + b)(2 \times 3 \times 2 + (-1)) \Rightarrow (2ax + b)(2 \times 6 + (-1)) \Rightarrow (2ax + b)(12 + (-1))$ $\Rightarrow (2ax + b) \times 11 \Rightarrow (2ax + 2) \times 11 \Rightarrow (2a \times 2 + 2) \times 11 \Rightarrow (2 \times 3 \times 2 + 2) \times 11 \Rightarrow (2 \times 6 + 2) \times 11$ $\Rightarrow (12 + 2) \times 11 \Rightarrow 14 \times 11 \Rightarrow 154$

"Random" evaluation of $(2ax + b)(2ax + c)$

$(2ax + b)(2ax + c) \Rightarrow 154$provided that $a \Rightarrow 3$, $b \Rightarrow 2$, $c \Rightarrow -1$ and $x \Rightarrow 2 \Rightarrow (2ax + b)(2 \times 3 \times x + c)$ $\Rightarrow (2ax + b)(6 \times x + c) \Rightarrow (2 \times 3 \times x + b)(6 \times x + c) \Rightarrow (2 \times 3 \times 2 + b)(6 \times x + c) \Rightarrow (2 \times 6 + b)(6 \times x + c)$ $\Rightarrow (2 \times 6 + b)(6 \times 2 + c) \Rightarrow (2 \times 6 + b)(12 + c) \Rightarrow (2 \times 6 + b)(12 + (-1)) \Rightarrow (12 + b)(12 + (-1))$ $\Rightarrow (12 + 2)(12 + (-1)) \Rightarrow 14 \times (12 + (-1)) \Rightarrow 14 \times 11 \Rightarrow 154$

1.2 Referential transparency

Referential transparency means that two identical λ-expressions on different places will have the same value if they are placed in equivalent context.

Referential transparency

- $(2ax + b)(2ax + c) \Rightarrow 154$ provided that $a \Rightarrow 3$, $b \Rightarrow 2$, $c \Rightarrow -1$ and $x \Rightarrow 2 \Rightarrow (2 \times 3 \times 2 + b)(2ax + c) \Rightarrow (12 + b)(2ax + c) \Rightarrow (12 + b)(12 + c) \Rightarrow (12 + 2)(12 + c) \Rightarrow 14 \times (12 + c) \Rightarrow 14 \times (12 + (-1)) \Rightarrow 14 \times 11 \Rightarrow 154$

1.3 Bound and free identifiers

Bound identifiers [1] are common in the world of mathematics.

$$\sum_{i=1}^{n}(i^2 + 1)$$

Here i is a bound identifier. It is a characteristic of bound variables that they can be systematically changed without altering the meaning (the value) of a formula. For example,

$$\sum_{k=1}^{n}(k^2 + 1)$$

means exactly the same thing as the previous summation. Similarly, the integral of $x^2 - 3x$ with respect to x

$$\int_0^t x^2 - 3x\,dx$$

is the same as the integral of $u^2 - 3u$ with respect to u

$$\int_0^t u^2 - 3u\,du$$

In set theory, the set of all x such that $x \geq 0$ is the same as the set of all y such that $y \geq 0$

$$\{x \mid x \geq 0\} = \{y \mid y \geq 0\}$$

Examples from mathematics

$\sum_{i=1}^{n}(i^2 + 1)$	$\sum_{k=1}^{n}(k^2 + 1)$
$\int_0^t x^2 - 3x\,dx$	$\int_0^t u^2 - 3u\,du$
$\{x \mid x \geq 0\}$	$\{y \mid y \geq 0\}$
$\forall x(x + 1 > x)$	$\forall y(y + 1 > y)$

There are two important notions: *bound occurrence* and *free occurrence*. Consider the expression

$$i\sum_{j=1}^{n}(j^2+i-a)$$

Here j in j^2+i-a is called bound occurrence of the identifier j. It is bound by summation operator $\sum$, which is called *binding site* (or just *binding*) of this occurrence of j. Because j is bound, we can change it to any other identifier but i and a without changing the meaning of the expression:

$$i\sum_{k=1}^{n}(k^2+i-a)$$

Any occurrence of an identifier that is not a bound occurrence is called *free occurrence*. For example i, n and a all occur free in the previous expression. Clearly, if we change a free identifier we have changed the meaning of the expression

$$i\sum_{j=1}^{n}(j^2+m-a)$$

Related notions

- bound identifier is j
- bounding site is $\sum$
- free identifiers are i, n and a

$$i\sum_{j=1}^{n}(j^2+i-a)$$

2 λ-calculus programming language basics

2.1 Syntax

When bound identifiers are used in function definitions they are often called *formal parameters*. For example, in

$$f(x) \equiv x^2 - 3x$$

x is the bound identifier or formal parameter. In lambda calculus this function would be written as

$$\lambda x(x^2 - 3x)$$

which can be read "that function which takes any x into $x^2 - 3x$." The Greek letter λ does not stand for anything.

In the λ-calculus, function application is just a substitution process. For example, suppose we have defined

$$f \equiv \lambda x(x^2 - 3x)$$

and want to know the value of $f(5)$; we can find it by substituting 5 for x throughout its scope. More specifically, we start with

$$f(5)$$

Substituting $\lambda x(x^2 - 3x)$ for f, we get

$$\lambda x(x^2 - 3x)(5)$$

Now we replace this expression by a copy of the *body* of *f*, i.e.

$$(5)^2 - 3(5) \Rightarrow 25 - 15 \Rightarrow 10$$

Example λ expression

- *abstraction* describes some computation (function)

$$f(x) \equiv x^2 - 3x$$

$$\lambda x(x^2 - 3x)$$

- *substitution rule* is used when a function is applied to parameters

$$(f\,5) \Rightarrow (\lambda x(x^2 - 3x)\,5) \Rightarrow 5^2 - 3 \times 5 \Rightarrow 25 - 15 \Rightarrow 10$$

Pure λ-expressions are formed only by the following three rules:[1]

Formal Grammar

$$\text{expression} = \left\{ \begin{array}{l} \text{identifier} \\ (\lambda\,\text{identifier expression}) \\ (\text{expression expression}) \end{array} \right\}$$

Further contracts:

[1] This grammar would be ambiguous if there were allowed also identifiers consisting from multiple characters. Here, spaces could be inserted whereever this is advantageous (for the sake of readability).

- The other-most parentheses could be dropped.
- The application operation is associative from left to right, so we can write $(xy)z$ as xyz.

An expression λxE may be reduced by renaming to an expression λyF, where F is obtained from E by replacing all free occurrences of x in E by y. This is allowed only if y does not occur in E.

Substitution rule

The second rule (besides *substitution*) is *renaming*

$$\lambda x(x^2 + 2x + 1) \Rightarrow \lambda u(u^2 + 2u + 1)$$

$$\lambda xx \Rightarrow \lambda aa \Rightarrow \lambda gg \Rightarrow \lambda ff$$

$$\lambda x(\lambda y(xy)) \Rightarrow \lambda f(\lambda y(fy)) \Rightarrow \lambda f(\lambda a(fa))$$

$$\lambda x(x + 1) \Rightarrow \lambda u(u + 1)$$

$$\lambda x(\lambda y(x + y)) \Rightarrow \lambda a(\lambda y(a + y)) \Rightarrow \lambda x(\lambda b(a + b))$$

$$\lambda x(x + y) \not\Rightarrow \lambda y(y + y)$$

$$(\lambda x(\lambda y(x + y)))\ (y + 1) \not\Rightarrow \lambda y(y + 1 + y)$$

$$(\lambda x(\lambda y(x + y)))\ (y + 1) \Rightarrow (\lambda x(\lambda z(x + z)))\ (y + 1) \Rightarrow \lambda x(y + 1 + z)$$

In λ-calculus we have a similar problem with substitution as we had in the land of logic [7, §3.1.3] [8, §3.3]. If we have an application:

$$(\lambda xE)\ F$$

i.e. two λ-expressions: λxE and F where λxE is applied to F which is its parameter. We say that F is substitutable instead of x in E if[2] F does not contain free occurrences of x. If it contained free occurrence of x, it would be captured (become bound) performing the substitution. Therefore the following λ-expression:

$$(\lambda x(\lambda y(x + y)))\ (y + 1)$$

cannot be naïvely reduced to:

$$\lambda y(y + 1 + y)$$

[2] Actually, F shouldn't contain any free variables which could after substitution become bound.

because the latter λ-expression would have different value than the original one. The substitution should be performed in two steps. First we rename the problematic bound variable:

$$(\lambda x(\lambda z(x + z)))\ (y + 1)$$

and then we perform the actual substitution:

$$\lambda z(y + 1 + z)$$

To make this affair even clearer we give another example from (perhaps) more obvious calculus:

Consider an expression:

$$\sum_{i=10}^{100} i + x$$

Suppose that we have learned that

$$x = y + 1$$

We can substitute free occurrences of the variable x in expression $\sum_{i=10}^{100} i + x$ with $y + 1$ as follows:

$$\sum_{i=10}^{100} i + y + 1$$

We are sure that the latter expression is equal to the original one (of course under that assumption that $x = y + 1$).

Now suppose that we have revealed that:

$$x = i + 1$$

Do you think that we could naïvely substitute $i + 1$ instead of x in the original expression:

$$\sum_{i=10}^{100} i + x$$

? Obviously not because if we did that we would get

$$\sum_{i=10}^{100} i + i + 1$$

and that is not equal to the original expression even under the assumption that $x = i + 1$. In order to be able to make that substitution we have first to rename the problematic bound variable to some *fresh variable* for example j:

$$\sum_{j=10}^{100} j + x$$

Now we are free to perform the substitution $i + 1$ instead of all free occurrences of x. This yields:

$$\sum_{j=10}^{100} j + i + 1$$

The last expression is equal to the original one under the assumption that $x = i + 1$.

2.2 Normal form

If and when an expression is reduced to the extent that the substitution rule can no longer be applied, it is said to be in *normal form* (sometimes *reduced* form). Intuitively, an expression is in normal form when it is an answer (i.e., it is done computing).

Normal form

definition

Not Normal	**Normal**
$(\lambda xx)\ y$	y
$(\lambda y(fy))\ a$	fa
$(\lambda x(\lambda y(xy)))\ a$	$\lambda y(ay)$
$(\lambda xx)\ (\lambda xx)$	λxx
$(\lambda x(xx))\ y$	yy
$(\lambda x(xy))\ (\lambda x(xx))$	yy

There are λ-expressions which do not have normal form. The *Church-Roser theorem* says, however, that if it is possible to reduce a given λ-expression to its normal form then we always get the same expression regardless of the order in which we perform reductions of subexpressions.

Non-normalizable λ-expressions

There are λ-expressions which do not have normal form.

$$
\begin{aligned}
(\lambda x(xx))\ (\lambda x(xx)) &\Rightarrow (\lambda x(xx))\ (\lambda x(xx)) \\
&\Rightarrow (\lambda x(xx))\ (\lambda x(xx)) \\
&\Rightarrow (\lambda x(xx))\ (\lambda x(xx)) \\
&\Rightarrow \ldots
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x(f(xx)))\ (\lambda x(f(xx))) &\Rightarrow f((\lambda x(f(xx)))\ (\lambda x(f(xx)))) \\
&\Rightarrow f(f((\lambda x(f(xx)))\ (\lambda x(f(xx))))) \\
&\Rightarrow f(f(f((\lambda x(f(xx)))\ (\lambda x(f(xx)))))) \\
&\Rightarrow \ldots
\end{aligned}
$$

Church-Roser theorem [1, §9]

2.3 Multiple Parameters

We argue that the λ-calculus is *computationally complete*, that is, that it can compute any computable function. In one sense this is trivial since the notion of computability is often defined in reference to the λ-calculus. On the other hand, it's certainly not intuitively obvious that the pure λ-calculus contains the necessary power to write all programs. Therefore here we will define in the pure λ-calculus the basic data types commonly used in programs: booleans, integers, tuples and sequences.

Although the λ-expression we have defined to have only one parameter, we can get the effect of two parameters by nesting functional abstractions, a technique known as *currying*. For example,

$$(\lambda x(\lambda y(x+y)))\ 3\ 1 \Rightarrow \lambda y(3+y)\ 1 \Rightarrow 3+1 \Rightarrow 4$$

Because such curried functions are so common in the λ-calculus, we allow the following abbreviation (also using '$\Rightarrow$' as a sign for abbreviation):

$$\lambda xy.(x+y) \Rightarrow \lambda x\,[\lambda y\,(x+y)]$$

If we adopt the convention that the body (the expression after the dot) extends as far to the right as possible, this enables us sometimes to drop parentheses as below:

$$\lambda xy.x+y$$

The resulting λ-expressions are more concise. Of course, it is possible to use dot also with single parameter lambda expressions:

$$\lambda x.x^2+2x+1$$

is the same as:

$$\lambda x(x^2+2x+1)$$

Currying

Currying can help us to create λ-expression with multiple parameters:

$$\lambda x\,[\lambda y\,(x+y)]$$

If you apply two numbers to this lambda expression:

$$\lambda x\,[\lambda y\,(x+y)]\,3\,1 \Rightarrow \lambda y(3+y)\,1 \Rightarrow 3+1 \Rightarrow 4$$

$$\lambda xy.(x+y) \Rightarrow \lambda x\,[\lambda y\,(x+y)]$$

$$\lambda xy.x+y \Rightarrow \lambda x\,[\lambda y\,(x+y)]$$

$$(\lambda abc.\,ax^2+bx+c)\,9\,6\,1 \Rightarrow 9x^2+6x+1$$

So the *dot convention* is only abbreviation, it can be directly translated to pure λ-calculus. It is here only for our convenience. Things revealed about pure λ-calculus hold also for the calculus with extended syntax—and vice versa.

Currying

General rules:

$$\lambda x_1 x_2 \dots x_n.E \Rightarrow \lambda x_1\{\lambda x_2[\dots(\lambda x_n E)\dots]\}$$

or, using the dot convention:

$$\lambda x_1 x_2 \dots x_n.E \Rightarrow \lambda x_1.\lambda x_2 \dots \lambda x_n E$$

2.4 Abbreviations

Abbreviations (or syntactic suggar or macros) do not extend the original power of the core language. They only introduce new (higher-level) constructs which can be, however, translated to the more primitive constructs which in turns can be translated into more primitive constructs and so on ultimately into the original core language. Abbreviations enable us to think in larger blocks.

They are very common in mathematics. Consider for example definition [8] of the macro

$$\text{dom } R$$

which gives us the one of the two projections of the binary relation R. It is defined as follows:

$$\text{dom } R \equiv \{x : X; y : Y \mid x \mapsto y \in R \bullet x\}$$

If we have this definition, we know how to expand the expression like the one on the left of the definition to the expression on the right in the definition.

λ-expression can be quite large. To be able to program significant functions in λ-calculus, it is convenient to have a way to attach names to λ-expressions. Therefore we allow rewrite rules of the following form:

Abbreviations

- We can give names to λ-expressions.

$$\begin{aligned} \text{plusp} &\Rightarrow \lambda x.\, x \geq 0 \\ \text{minusp} &\Rightarrow \lambda x.\, x < 0 \\ \text{succ} &\Rightarrow \lambda x.x + 1 \\ \text{square} &\Rightarrow \lambda x.x^2 \end{aligned}$$

- reasons

- semantics

$$\text{minusp}\,(\text{succ}\,2) \Rightarrow (\lambda x.\, x < 0)(\text{succ}\,2) \Rightarrow (\lambda x.\, x < 0)[\lambda x x\, x + 1]\, 2]$$

These rules simply say, for example, that 'plusp' is an abbreviation for λ-expression '$\lambda x.\, x \geq 0$'. It must be emphasized that these rewrite rules are not part of the λ-calculus, but simply abbreviations that *we* use for talking *about* λ-expressions. Abbreviations introduced in this way can always be eliminated by applying the rewrite rules until no abbreviations are left. As long as this is understood, however, there is no reason that we can't use rules introduced above directly in reductions, as in the following example:

Abbreviations

$$\begin{aligned}
\text{minusp}\,(\text{succ}\,2) &\Rightarrow (\lambda x.\, x < 0)(\text{succ}\,2) \\
&\Rightarrow (\text{succ}\,2) < 0 \\
&\Rightarrow (\lambda x.\, x + 1)2 < 0 \\
&\Rightarrow 2 + 1 < 0 \\
&\Rightarrow 3 < 0 \\
&\Rightarrow \textbf{false}
\end{aligned}$$

Abbreviations and syntactic sugar could be regarded as macros which ultimately expand to pure λ-expressions. For example the *plusp* above could be regarded as a macro (introduced by us for convenience) which expands to $\lambda x.x \geq 0$.

3 Basic data-types

This section introduces various basic data-types to intuitively persuade reader that λ-calculus is complete—i.e. it can be indeed used to write any algorithm.

3.1 Booleans

Booleans

We would like to use booleans (**true** and **false**) for branching as follows:

$$\begin{aligned}
\textbf{true}\ t\ f &\Rightarrow t \\
\textbf{false}\ t\ f &\Rightarrow f
\end{aligned}$$

This leads us immediately to the following definitions:

$$\begin{aligned}\textbf{true} &\Rightarrow \lambda\, tf.\, t\\ \textbf{false} &\Rightarrow \lambda\, tf.\, f\end{aligned}$$

if **and** not **operation**

Function enabling us to branch could be defined as:

$$\text{if} \Rightarrow \lambda ctf.\, ctf$$

We can check if indeed it behaves as it should behave:

$$\begin{aligned}\text{if}\,\textbf{true}\; a\, b &\Rightarrow (\lambda ctf.\, ctf)\;\textbf{true}\; a\, b\\ &\Rightarrow \textbf{true}\; a\, b\\ &\Rightarrow a\end{aligned}$$

Function ‘not’ should behave as follows:

$$\begin{aligned}\text{not}\;\textbf{true} &\Rightarrow \textbf{false}\\ \text{not}\;\textbf{false} &\Rightarrow \textbf{true}\end{aligned}$$

This suggests the following definition:

$$\text{not} \Rightarrow \lambda x\,.\,\text{if}\; x\;\textbf{false}\;\textbf{true}$$

and **operation**

Looking at the desired behavior of the and operation:

$$\begin{aligned}\text{and}\,\textbf{true}\,\textbf{true} &\Rightarrow \textbf{true}\\ \text{and}\,\textbf{true}\,\textbf{false} &\Rightarrow \textbf{false}\\ \text{and}\,\textbf{false}\,\textbf{true} &\Rightarrow \textbf{false}\\ \text{and}\,\textbf{false}\,\textbf{false} &\Rightarrow \textbf{false}\end{aligned}$$

we can infer definition of the and operation as:

$$\text{and} \Rightarrow \lambda xy.\,\text{if}\; x\; y\,\textbf{false}$$

or **operation**

Would you be able to guess the a definition of the or operation if it behaves as follows:

$$\begin{aligned} \text{or}\,\textbf{true}\,\textbf{true} &\Rightarrow \textbf{true} \\ \text{or}\,\textbf{true}\,\textbf{false} &\Rightarrow \textbf{true} \\ \text{or}\,\textbf{false}\,\textbf{true} &\Rightarrow \textbf{true} \\ \text{or}\,\textbf{false}\,\textbf{false} &\Rightarrow \textbf{false} \end{aligned}$$

?

or **operation**

$$\text{or} \Rightarrow \lambda xy.\, \text{if}\, x\, true\, y$$

equal **operation**

Since now we have complete system of logical connective, we can easily define other common logical connectives:

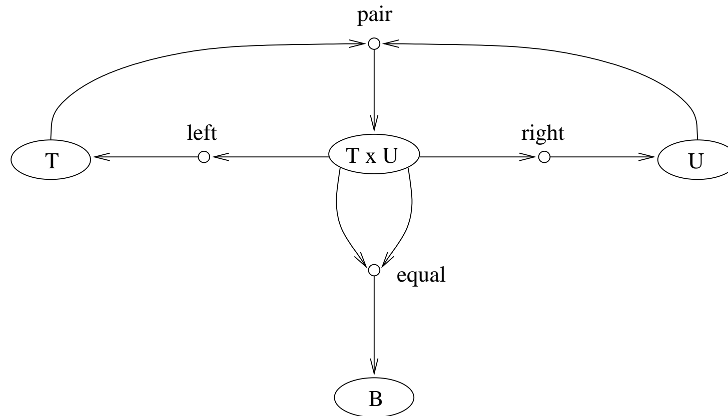
$$\begin{aligned} \text{imp} &\Rightarrow \lambda xy.\, \text{or}\,(\text{not}\, x)\, y \\ \text{nand} &\Rightarrow \lambda xy.\, \text{not}\,(\text{and}\, x\, y) \\ \text{nor} &\Rightarrow \lambda xy.\, \text{not}\,(\text{or}\, x\, y) \\ \text{equal} &\Rightarrow \lambda xy.\, \text{or}\,[\text{and}\, x\, y]\,[\text{and}\,(\text{not}\, x)\,(\text{not}\, y)] \end{aligned}$$

There are no problems with booleans.

3.2 Direct Product Types

We must find a way to represent tuples so that all the selectors and constructors of direct product data types can be implemented. Since the λ-calculus is a calculus of functions, tuples must ultimately be represented as functions of some form. Furthermore, our definition of tuples must be based on the the facilities already defined: the pure λ-calculus and Booleans. How can these help us? Recall that the most basic property of the selectors and constructors of tuples is that they invert each other. Therefore letting pair $x\,y$ represent the (curried) operation that constructs the pair (x, y), we have

Direct Product of Types



"Specification"

Recall that the most basic property of the *selectors* and *constructors* of tuples is that they invert each other.

$$\begin{aligned} \text{left}\,(\text{pair}\,x\,y) &= x \\ \text{right}\,(\text{pair}\,x\,y) &= y \end{aligned}$$

That is, the *pair* constructor must construct a pair in such a way that it is possible to recover each of the two elements that went into the pair. Thus we need a way to select either of these two elements. Our Boolean data type provides exactly this capability, since $\mathbf{true}\,x\,y \Rightarrow x$ and $\mathbf{false}\,x\,y \Rightarrow y$. That is, $(c\,x\,y)$ selects either x or y from the pair depending on whether c is **true** or **false**. Therefore lets define the pair returned by $\text{pair}\,x\,y$ to be a function that can return either x or y depending on whether it is applied to **true** or to **false**:

$$\text{pair}\,x\,y \quad \Rightarrow \quad \lambda c.cxy$$

For example, the tuple (1, **true**) is represented by the λ-expression $\lambda c.c\,1\,\mathbf{true}$. Hence the prototype implementation of pair, left and right are: This leads us to:

Prototype implementation

$$\begin{aligned} \text{pair}\,x\,y &\Rightarrow \lambda xy.\lambda c.cxy \\ \text{left} &\Rightarrow \lambda x.x\,\mathbf{true} \\ \text{right} &\Rightarrow \lambda x.x\,\mathbf{false} \end{aligned}$$

Example:

$$
\begin{aligned}
\text{left}\,(\text{pair}\ A\,B) &\Rightarrow \text{left}\,[(\lambda xy.\lambda c.cxy)\ A\,B] \\
&\Rightarrow \text{left}\,(\lambda c.cAB) \\
&\Rightarrow (\lambda x.x\,\mathbf{true})(\lambda c.cAB) \\
&\Rightarrow (\lambda c.cAB)\,\mathbf{true} \\
&\Rightarrow \mathbf{true}\,A\,B \\
&\Rightarrow A
\end{aligned}
$$

Exercise 8.15

Develop a prototype implementation of the equal operation for tuples: equal $x\,y$ is **true** if and only if x and y are the same pairs. Assume that you have operations equal_σ and equal_τ applicable to the component types of the tuple.

$$\text{equal} \Rightarrow \lambda xy.\ \ldots$$

Solution of Exercise 8.15

$$
\begin{aligned}
\text{equal} \Rightarrow \lambda xy.\text{and}\,&(\text{equal}_\sigma\,(\text{left}\,x)(\text{left}\,y)) \\
&(\text{equal}_\tau\,(\text{right}\,x)(\text{right}\,y))
\end{aligned}
$$

Now I a bit fork from the original text of the book. On pages 376–377 they introduce new syntactic abbreviation which enable them to to define new lambda expressions of the following form:

$$\max \Rightarrow \lambda\,(x,y).\text{if}\,(x<y)\,y\,x$$

This is not a pure λ-expression because hitherto we only allowed ordinary identifiers to be formal parameters of λ-expressions. It is not at all difficult to introduce this syntactic sugar but under the time constrains we decided not to do so. Instead, where in the original text occur expression with this syntactic sugar, we present them here in expanded (sugar-free) form. We hope that until now the things were quite clear and we fear that with this obscurity things would not be that clear.

Further syntactic abbreviations

We skip here introduction of the following syntactic sugar:

$$\lambda\,(x, y) \Rightarrow E(x, y)$$

example

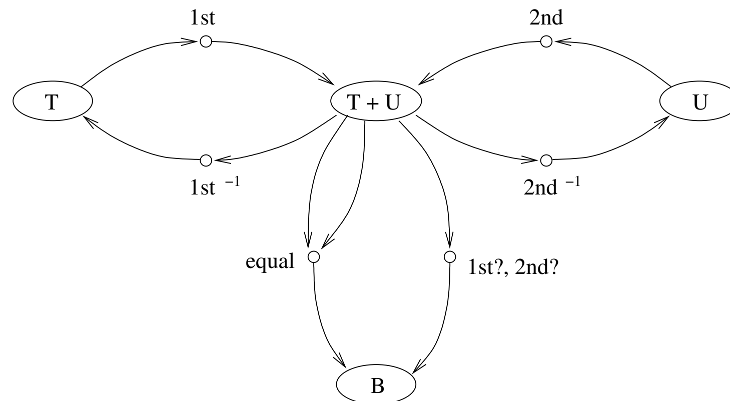
$$\max \Rightarrow \lambda\,(x, y).\, \text{if}\,(x > y)\, x\, y$$

The above expression is not a pure λ-expression, but we can translate it to pure λ-expression. When we show how to translate any similar expression to pure λ-calculus we could adopt the above construct to our language while technically using the same old language (but maybe with a bit more convenient syntax). We will skip introduction of this construct. It would take a time, we do not have time and if it is not grasped properly, one can easily loose track of things.

3.3 Direct Sum of Types

Sometimes it is useful to represent the members of what is intuitively one abstract type by values drawn from two or more concrete types (i.e. to make some kind of union of two types).

Direct Sum of Types



Prototype Implementation

constructors, selectors, predicates

$$\begin{aligned}
\text{1st} &\Rightarrow \lambda x.\, \text{pair}\, \mathbf{true}\, x \\
\text{2nd} &\Rightarrow \lambda y.\, \text{pair}\, \mathbf{false}\, y \\
\text{1st}^{-1} &\Rightarrow \lambda p.\, \text{right}\, p \\
\text{2nd}^{-1} &\Rightarrow \lambda p.\, \text{right}\, p \\
\text{1st?} &\Rightarrow \lambda p.\, \text{left}\, p \\
\text{2nd?} &\Rightarrow \lambda p.\, \text{not 1st?}\, p
\end{aligned}$$

The last operation could be more effectively expressed as:

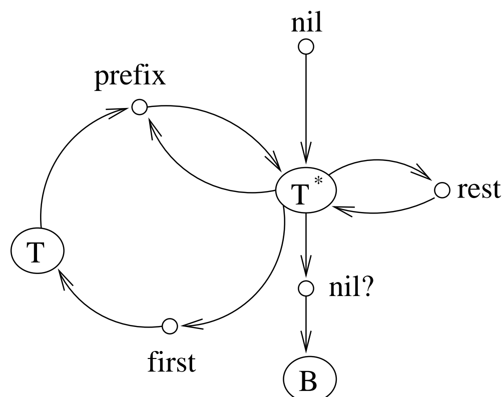
$$\text{2nd?} \Rightarrow \lambda p.\, \text{not left}\, p$$

3.4 Sequences

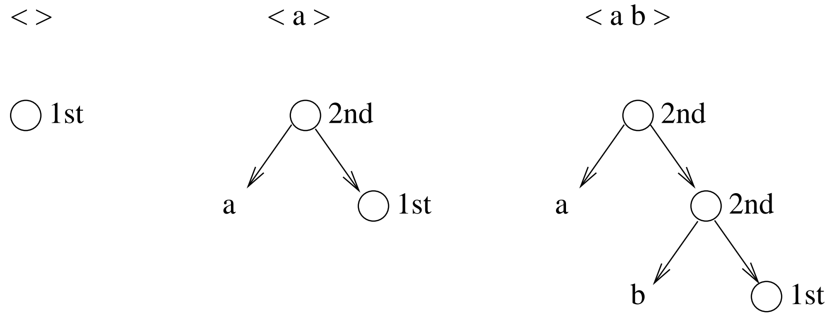
We can use direct product and direct sum of types for realizing prototype of sequences. Direct product (i.e. couples) will be used for realizing “cons-cells”. Direct sum will be used for uniting “anything meaningful” (of the first type) with “nil” (the only element of the second type). Sequences could be defined as recursive data types [1, §5.3].

Signature of Sequences

constructors, selectors, predicates



Sketch of Realization



Prototype Implementation

constructors, selectors, predicates

$$
\begin{array}{rcl}
\text{nil} & \Rightarrow & \text{1st}\ \mathbf{true} \\
\text{prefix} & \Rightarrow & \lambda xy.\, \text{2nd}\,(\text{pair}\, x\, y) \\
\text{first} & \Rightarrow & \lambda s.\, \text{left}\,(\text{2nd}^{-1}\, s) \\
\text{rest} & \Rightarrow & \lambda s.\, \text{right}\,(\text{2nd}^{-1}\, s) \\
\text{nil?} & \Rightarrow & \text{1st?}
\end{array}
$$

Note that within the nil constructor could be just about any value. Eventually also the current one—**true**.

Examples

$$
\begin{array}{rcl}
\text{nil? nil} & \Rightarrow & \text{1st? nil} \\
 & \Rightarrow & (\lambda p.\text{left}\ p)\,\text{nil} \\
 & \Rightarrow & \text{left nil} \\
 & \Rightarrow & \text{left}\ (\text{1st}\ \mathbf{true}) \\
 & \Rightarrow & \text{left}\ ((\lambda x.\text{pair}\ \mathbf{true}\ x)\ \mathbf{true}) \\
 & \Rightarrow & \text{left}\ (\text{pair}\ \mathbf{true}\ \mathbf{true}) \\
 & \Rightarrow & \mathbf{true}
\end{array}
$$

Some steps where omitted.

Additional Syntactic Sugar

We could enrich the syntax of our language so that:

$\langle\rangle$	nil
$\langle a\rangle$	prefix a, nil
$\langle a, b\rangle$	prefix a, (prefix b, nil)
$\langle a, b, c\rangle$	prefix a, (prefix b, (prefix c, nil))
	...

Examples

$$\begin{aligned} \text{first } \langle a,b,c\rangle &\Rightarrow a \\ \text{rest } \langle a,b,c\rangle &\Rightarrow \langle b,c\rangle \\ \text{first (rest } \langle a,b,c\rangle) &\Rightarrow b \end{aligned}$$

Some steps were omitted.

4 Auxiliary Declarations

It makes no sense to ask the value of an *open expression*, that is, an expression containing free variables. Therefore any complete functional program must be *closed*: All its identifiers must be bound.

Auxiliary Declarations

- it does not have sense to ask of an *open expression*
- each functional program must therefore be *closed*

When E is an expression with a single free variable and we want to evaluate it in a context where v has value x we can do it as follows:

$$(\lambda v E)x$$

In order to be able to express that clearly, dedicated syntactic sugar was introduced:

$$\textbf{let } v \equiv x \textbf{ in } E \Rightarrow (\lambda v E)x$$

Auxiliary Declarations in General

$$\textbf{let } v_1 \equiv x_1 \textbf{ and } v_2 \equiv x_2 \textbf{ and } \ldots \textbf{ and } v_n \equiv x_n \textbf{ in } E \Rightarrow$$
$$\Rightarrow (\lambda v_1, v_2, \ldots, v_n.E)\, x_1 x_2 \ldots x_n$$

Evaluating an expression with a single free variable:

$$\begin{aligned}\textbf{let } x \equiv 10 \textbf{ in } x + 10 &\Rightarrow (\lambda x(x+10))\, 5 \\ &\Rightarrow 5 + 10 \\ &\Rightarrow 15\end{aligned}$$

Evaluating an expression with two free variables:

$$\begin{aligned}\textbf{let } x \equiv 10 \textbf{ and } y \equiv 20 \textbf{ in } x + y &\Rightarrow (\lambda xy.(x+y))\ 10\ 20 \\ &\Rightarrow 10 + 20 \\ &\Rightarrow 30\end{aligned}$$

5 Recursive Declarations

Our prototype definition of **let** does not permit recursive declarations.

The scope of 'fac' is 'fac 4', as can be seen by eliminating the syntactic sugar.

Recursive Declarations

How to define recursive functions?

How could we, for example, define function which computes the factorial of a given integer?

The first attempt:

$$\lambda n.\text{if}[n = 0, 1, n \times \text{fac}(n-1)]$$

The second attempt:

$$\textbf{let } \text{fac} \equiv \lambda n.\text{if}[n = 0, 1, n \times \text{fac}(n-1)] \textbf{ in } \text{fac } 4$$

This seems promising until we try to reduce it:

$$\Rightarrow (\lambda\, \text{fac.fac } 4)\ \ \{\lambda n.\text{if}[n = 0, 1, n \times \text{fac}(n-1)]\}$$

There we see that we have a unbound variable fac there.

If in some imperative programming language we wanted to perform a given command S over and over until some condition is met and we only had branching, we would need "infinite" flow-charts:

Recursive Declarations

Recursion in λ-calculus must be expressed completely differently because here we have no global function names. Only λ-expressions.

It seems as if we (in order to define a recursive function) needed λ-expression of infinite length:

$\varphi \equiv \{\lambda n.\text{if}\,[n = 0,\, 1,\, n\times \quad \{\lambda n.\text{if}\,[n = 0,\, 1,\, n\times \quad \{\lambda n.\text{if}\,[n = 0,\, 1,\, n\times \,\ldots\, (n-1)]\}(n-1)]\}(n-1)]\}$

It is possible to observe certain some kind of “copying pattern” there.

An infinite formula is not much use to us because we cannot write it down; we can only write down approximations of it. For example, the following are approximations of the formula φ:

Recursive Declarations

The “copying pattern”:

$$
\begin{aligned}
\varphi_{-1} &\equiv \bot \\
\varphi_0 &\equiv \lambda n.\, \text{if}\,[n = 0,\, 1,\, n \times \varphi_{-1}(n-1)] \\
\varphi_1 &\equiv \lambda n.\, \text{if}\,[n = 0,\, 1,\, n \times \varphi_{0}(n-1)] \\
\varphi_2 &\equiv \lambda n.\, \text{if}\,[n = 0,\, 1,\, n \times \varphi_{1}(n-1)] \\
\varphi_3 &\equiv \lambda n.\, \text{if}\,[n = 0,\, 1,\, n \times \varphi_{2}(n-1)] \\
\varphi_4 &\equiv \lambda n.\, \text{if}\,[n = 0,\, 1,\, n \times \varphi_{3}(n-1)] \\
&\vdots \\
\varphi_\infty &\equiv \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{aligned}
$$

- φ_{-1} is not able to compute factorial of any integer.
- φ_0 is able to compute factorial of a single integer 0.
- φ_1 is able to compute factorial of integers 0 and 1.
- φ_2 is able to compute factorial of integers 0, 1 and 2.
- φ_3 is able to compute factorial of integers 0, 1, 2 and 3.
- φ_4 is able to compute factorial of integers 0, 1, 2, 3 and 4.
- ...
- φ_∞ (a λ-expression of infinite length) would be able to compute factorial of any integer.

But, "infinite" flow-charts would need that our source code would have an infinite number of bytes and that wouldn't be compilable. Each program should have finite source-code (or flow-chart or some analogous representation). Fortunately, in imperative programming we have cycles which eliminate the need for (possibly) infinite programs.

However, recall that there were so called "non-normalizable" λ-expressions. One of them

$$[\lambda x.F(xx)][\lambda x.F(xx)]$$

is extremely interesting because it seem to "unfold forever".

Recursive Declarations

The only way out is to use some kind of λ-expression which unfolds forever. We have already some λ-expressions of that sort:

$$[\lambda x(xx)][\lambda x(xx)]$$

and

$$[\lambda x.f(xx)][\lambda x.f(xx)]$$

Let's close the second λ-expression above as follows:

$$\mathbf{Y} \Rightarrow \lambda f.\,[\lambda x.f(xx)][\lambda x.f(xx)]$$

and thus:

$$\mathbf{Y}f \Rightarrow [\lambda x.f(xx)][\lambda x.f(xx)]$$

One of its properties is that it is possible to expand expression $\mathbf{Y}f$ it in the following way:

$$\mathbf{Y}f \Rightarrow f(\mathbf{Y}f) \Rightarrow f(f(\mathbf{Y}f)) \Rightarrow f(f(f(\mathbf{Y}f)))$$

Recursive Declarations

$$\mathbf{Y}f \Rightarrow f(\mathbf{Y}f) \Rightarrow f(f(\mathbf{Y}f)) \Rightarrow f(f(f(\mathbf{Y}f)))$$

Detailed reduction:

$$\begin{aligned}
\mathbf{Y}f &\Rightarrow [\lambda x.f(xx)][\lambda x.f(xx)] \\
&\Rightarrow f\ \{[\lambda x.f(xx)][\lambda x.f(xx)]\} \\
&\Rightarrow f\ (f\ \{[\lambda x.f(xx)][\lambda x.f(xx)]\}) \\
&\Rightarrow f\ (f\ (f\ \{[\lambda x.f(xx)][\lambda x.f(xx)]\}))
\end{aligned}$$

This way, we are able to express certain kind of "infinite expressions".

Recursive Declarations

Now we can define recursive functions as follows:

$$\begin{array}{lcl} \text{fac} & \Rightarrow & \mathbf{Y}\,\{\lambda f.\lambda n.\text{if}\,[n=0,1,n\times f(n-1)]\} \\ \text{fib} & \Rightarrow & \mathbf{Y}\,\{\lambda f.\lambda n.\text{if}\,[or(n=0,n=1),1,f(n-1)+f(n-2)]\} \end{array}$$

Recursive Declarations

Suppose that:

$$\psi \Rightarrow \lambda f.\lambda n.\text{if}\,(n=0,1,n\times f(n-1))$$

Computation of a factorial of number 0 (zero):

$$\begin{array}{lcl} \text{fac } 0 & \Rightarrow & \mathbf{Y}\psi\,0 \Rightarrow \\ & \Rightarrow & \psi(\mathbf{Y}\psi)\ 0 \Rightarrow \\ & \Rightarrow & \lambda n.\text{if}\,(n=0,1,n\times \mathbf{Y}\psi(n-1))\ 0 \Rightarrow \\ & \Rightarrow & \text{if}\,(0=0,1,0\times \mathbf{Y}\psi(0-1)) \Rightarrow \\ & \Rightarrow & 1 \end{array}$$

Recursive Declarations

Suppose that:

$$\psi \Rightarrow \lambda f.\lambda n.\text{if}\,(n=0,1,n\times f(n-1))$$

Computation of a factorial of number 1 (one):

$$\begin{array}{lcl} \text{fac } 1 & \Rightarrow & \mathbf{Y}\psi\,1 \Rightarrow \\ & \Rightarrow & \psi(\mathbf{Y}\psi)\ 1 \Rightarrow \\ & \Rightarrow & \lambda n.\text{if}\,(n=0,1,n\times \mathbf{Y}\psi(n-1))\ 1 \Rightarrow \\ & \Rightarrow & \text{if}\,(1=0,1,1\times \mathbf{Y}\psi(1-1)) \Rightarrow \\ & \Rightarrow & 1\times \mathbf{Y}\psi(1-1) \Rightarrow \\ & \Rightarrow & 1\times \mathbf{Y}\psi\,0 \Rightarrow \\ & \Rightarrow & 1\times 1 \Rightarrow \\ & \Rightarrow & 1 \end{array}$$

We have reused the fact that we already know that $\mathbf{Y}\psi\,0 \Rightarrow 1$. This made the actual reduction shorter.

Recursive Declarations

Suppose that:

$$\psi \Rightarrow \lambda f.\lambda n.\text{if}\,(n = 0, 1, n \times f(n-1))$$

Computation of a factorial of number 2 (two):

$$\begin{aligned}
\text{fac } 2 &\Rightarrow \mathbf{Y}\psi\,2 \Rightarrow \\
&\Rightarrow \psi(\mathbf{Y}\psi)\ 2\ \Rightarrow \\
&\Rightarrow \lambda n.\text{if}\,(n = 0, 1, n \times \mathbf{Y}\psi(n-1))\ 2 \Rightarrow \\
&\Rightarrow \text{if}\,(2 = 0, 1, 2 \times \mathbf{Y}\psi(2-1)) \Rightarrow \\
&\Rightarrow 2 \times \mathbf{Y}\psi(2-1) \Rightarrow \\
&\Rightarrow 2 \times \mathbf{Y}\psi\,1 \Rightarrow \\
&\Rightarrow 2 \times 1 \Rightarrow \\
&\Rightarrow 2
\end{aligned}$$

Again, we have reused the fact that we already know that $\mathbf{Y}\psi\,1 \Rightarrow 1$. This made the actual reduction shorter.

Recursive Declarations

Suppose that:

$$\psi \Rightarrow \lambda f.\lambda n.\text{if}\,(n = 0, 1, n \times f(n-1))$$

Computation of a factorial of number 3 (three):

$$\begin{aligned}
\text{fac } 3 &\Rightarrow \mathbf{Y}\psi\,3 \Rightarrow \\
&\Rightarrow \psi(\mathbf{Y}\psi)\ 3 \Rightarrow \\
&\Rightarrow \lambda n.\text{if}\,(n = 0, 1, n \times \mathbf{Y}\psi(n-1))\ 3 \Rightarrow \\
&\Rightarrow \text{if}\,(3 = 0, 1, 3 \times \mathbf{Y}\psi(3-1)) \Rightarrow \\
&\Rightarrow 3 \times \mathbf{Y}\psi(3-1) \Rightarrow \\
&\Rightarrow 3 \times \mathbf{Y}\psi\,2 \Rightarrow \\
&\Rightarrow 3 \times 2 \Rightarrow \\
&\Rightarrow 6
\end{aligned}$$

Again, we have reused the fact that we already know that $\mathbf{Y}\psi\,2 \Rightarrow 2$. This made the actual reduction shorter.

Recursive Declarations

Suppose that:

$$\psi \Rightarrow \lambda f.\lambda n.\text{if}\,(n = 0, 1, n \times f(n-1))$$

Computation of a factorial of number 4 (four):

$$\begin{aligned}
\text{fac } 4 &\Rightarrow \mathbf{Y}\psi\, 4 \Rightarrow \\
&\Rightarrow \psi(\mathbf{Y}\psi)\ 4 \Rightarrow \\
&\Rightarrow \lambda n.\text{if } (n = 0, 1, n \times \mathbf{Y}\psi(n-1))\ 4 \Rightarrow \\
&\Rightarrow \text{if } (4 = 0, 1, 4 \times \mathbf{Y}\psi(4-1)) \Rightarrow \\
&\Rightarrow 4 \times \mathbf{Y}\psi(4-1) \Rightarrow \\
&\Rightarrow 4 \times \mathbf{Y}\psi\, 3 \Rightarrow \\
&\Rightarrow 4 \times 6 \Rightarrow \\
&\Rightarrow 24
\end{aligned}$$

Again, we have reused the fact that we already know that $\mathbf{Y}\psi\, 3 \Rightarrow 6$. This made the actual reduction shorter.

Syntactic Sugar

If we have tools for expressing recursion, we can now make their utilization more convenient with:

$$\textbf{let rec } f \equiv x \textbf{ in } B \quad \Rightarrow \quad \textbf{let } f \equiv \mathbf{Y}(\lambda f.x) \textbf{ in } B$$

The following expression:

$$\textbf{let rec } \text{fac} \equiv \lambda n.\, \text{if } [n = 0, 1, n \times \text{fac}(n-1)] \textbf{ in } \text{fac } 4$$

expands to:

$$\textbf{let } \text{fac} \equiv \mathbf{Y}\ (\lambda\ \text{fac}.\lambda n.\, \text{if } [n = 0, 1, n \times \text{fac}(n-1)]) \textbf{ in } \text{fac } 4$$

and this expands to:

$$(\lambda\, \text{fac.fac } 4)\ (\mathbf{Y}\ (\lambda\ \text{fac}.\lambda n.\, \text{if } [n = 0, 1, n \times \text{fac}(n-1)]))$$

Evaluation Strategy

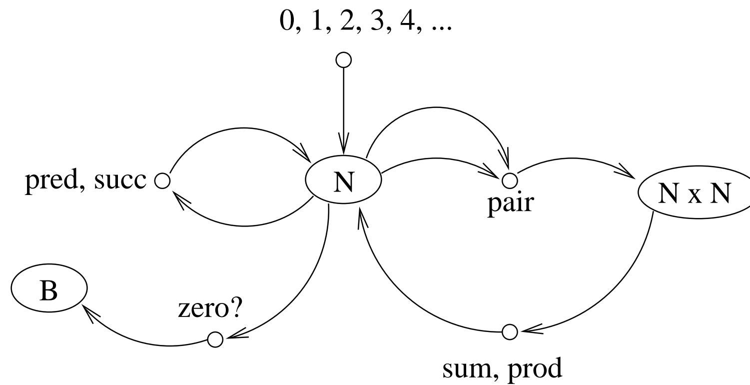
Evaluation stratey decides which of the possible applications will be applied next.

- eager
- random
- lazy

Sufficiently wrong strategy could cause that some expressions will never be reduced to normal form even if they have it.

6 Natural Numbers

Natural Numbers



Our goal is to show the computational completeness of the λ-calculus—in particular, that the primitives of a chosen concrete (pure) functional language can be implemented in the λ-calculus. For this purpose we do not need an efficient implementation; any correct implementation will do.

Natural Numbers

Integer 5 in unary notation:

| | | | |

or it could be:

* * * * *

The essence of unary notation is that a number is represented by example; to indicate five we show five things, with the understanding that the only relevant property is the number of things.

Since we have already defined sequences, we can translate these ideas directly into the λ-calculus. For example, the number five can be represented by a sequence of any five things. Just as in unary notation it doesn't really matter what these things are, so we can use any convenient value; here we use the value nil. Thus the number five is represented by a sequence of five nils:

$$\langle \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil} \rangle$$

The number zero, of course, is represented by the sequence containing no NILs, that is, the null sequence. This method works well for *natural numbers*.

Constructors, Operations and Zero Predicate

$$\begin{array}{rcl}
\underline{0} & \Rightarrow & \langle\rangle \\
\underline{1} & \Rightarrow & \langle \text{nil} \rangle \\
\underline{2} & \Rightarrow & \langle \text{nil, nil} \rangle \\
\underline{3} & \Rightarrow & \langle \text{nil, nil, nil} \rangle \\
\underline{4} & \Rightarrow & \langle \text{nil, nil, nil, nil} \rangle \\
 & \vdots & \\
\text{zero?} & \Rightarrow & \ldots \\
\text{succ} & \Rightarrow & \ldots \\
\text{pred} & \Rightarrow & \ldots
\end{array}$$

Constructors, Operations and Zero Predicate

$$\begin{array}{rcl}
\underline{0} & \Rightarrow & \langle\rangle \\
\underline{1} & \Rightarrow & \langle \text{nil} \rangle \\
\underline{2} & \Rightarrow & \langle \text{nil, nil} \rangle \\
\underline{3} & \Rightarrow & \langle \text{nil, nil, nil} \rangle \\
\underline{4} & \Rightarrow & \langle \text{nil, nil, nil, nil} \rangle \\
 & \vdots & \\
\text{zero?} & \Rightarrow & \lambda n.\, \text{nil?}\, n \\
\text{succ} & \Rightarrow & \ldots \\
\text{pred} & \Rightarrow & \ldots
\end{array}$$

Constructors, Operations and Zero Predicate

$$
\begin{aligned}
\underline{0} &\Rightarrow \langle\rangle \\
\underline{1} &\Rightarrow \langle \text{nil} \rangle \\
\underline{2} &\Rightarrow \langle \text{nil, nil} \rangle \\
\underline{3} &\Rightarrow \langle \text{nil, nil, nil} \rangle \\
\underline{4} &\Rightarrow \langle \text{nil, nil, nil, nil} \rangle \\
&\vdots \\
\text{zero?} &\Rightarrow \lambda n.\, \text{nil?}\, n \\
\text{succ} &\Rightarrow \lambda n.\, \text{prefix}\, \text{nil}\, n \\
\text{pred} &\Rightarrow \ldots
\end{aligned}
$$

Constructors, Operations and Zero Predicate

$$
\begin{aligned}
\underline{0} &\Rightarrow \langle\rangle \\
\underline{1} &\Rightarrow \langle \text{nil} \rangle \\
\underline{2} &\Rightarrow \langle \text{nil, nil} \rangle \\
\underline{3} &\Rightarrow \langle \text{nil, nil, nil} \rangle \\
\underline{4} &\Rightarrow \langle \text{nil, nil, nil, nil} \rangle \\
&\vdots \\
\text{zero?} &\Rightarrow \lambda n.\, \text{nil?}\, n \\
\text{succ} &\Rightarrow \lambda n.\, \text{prefix}\, \text{nil}\, n \\
\text{pred} &\Rightarrow \lambda n.\, \text{rest}\, n
\end{aligned}
$$

Constructors, Operations and Zero Predicate

$$
\begin{array}{rcl}
\underline{0} & \Rightarrow & \langle\rangle \\
\underline{1} & \Rightarrow & \langle \text{nil} \rangle \\
\underline{2} & \Rightarrow & \langle \text{nil, nil} \rangle \\
\underline{3} & \Rightarrow & \langle \text{nil, nil, nil} \rangle \\
\underline{4} & \Rightarrow & \langle \text{nil, nil, nil, nil} \rangle \\
 & \vdots & \\
\text{zero?} & \Rightarrow & \lambda n.\, \text{nil?}\, n \\
\text{succ} & \Rightarrow & \lambda n.\, \text{prefix}\, \text{nil}\, n \\
\text{pred} & \Rightarrow & \lambda n.\, \text{rest}\, n
\end{array}
$$

We could be more careful and define predecessor as:

$$\text{pred} \quad \Rightarrow \quad \lambda n.\, \text{if}\, (\text{zero?}\, n, \bot, \text{rest}\, n)$$

Addition

$$\textbf{let rec}\, \text{sum}\, (m, n) = \begin{cases} m & \text{if } n = 0 \\ \text{succ}\, (\text{sum}\, (m,\ \text{pred}\, n)) & \text{otherwise} \end{cases}$$

With less syntactic sugar:

$$\textbf{let rec}\, \text{sum}\, p = \begin{cases} \text{left}\, p & \text{if } (\text{right}\, p) = 0 \\ \text{succ}\, (\text{sum}\, (\text{left}\, p,\ \text{pred}\, (\text{right}\, p))) & \text{otherwise} \end{cases}$$

. . .

References

References

[1] B. J. Maclennan, Functional Programming: Practice and Theory, Addison-Wesley, 1990.

[2] Wikipedia.
URL `http://www.wikipedia.org`

[3] D. E. Knuth, The Art of Computer Programming: Fundamental Algorithms, Vol. 1, Addison Wesley, 1997.

[4] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[5] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[6] R. Milner, Communicating and Mobile Systems: The π-calculus, Cambridge University Press, 1999.

[7] V. Švejdar, LOGIKA, neúplnost, složitost a nutnost, Academia, nakladatelství České republiky, 2002.

[8] J. Woodcock, J. Davies, Using Z: specification, refinement, and proof, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
URL `http://www.usingz.com`