Lambda Calculus as a Programming Language

The Lambda calculus can be considered as the machine code of a particular computer. Call it the Lambda machine. As in conventional computers, layers of data representation and processing constructs with varying level of abstraction can be implemented on the Lambda machine starting from the basic machine code. In particular, these can be the basic elements of a weak-typed functional programming language. The elementary level of the language preserves the syntax of the Lambda calculus and defines a set of free variables as names of λ-expressions with predefined meaning that stand for values of various data types and for data type operators. A language that runs on a conventional machine adds new programming devices on an already rich base of primitive data such as numbers and Boolean values that are represented as strings of bits in the target machine. The arithmetic and the Boolean operators are also wired into the machine hardware. In the Lambda machine numbers and Boolean values are not predefined. They have to be represented as λ-expressions and their operators have to be defined from scrap. Instead of bits and binary operations here we have structured strings of symbols that designate functions and function applications. Finally, we could implement a programming environment based entirely on functions and function applications.

Besides enriching the calculus with meaningful operations and values specific to various data types, there are additional points of interest when viewing the Lambda calculus as a programming language. Some essential decisions address the following problems:

- The strategy of transferring function parameters: by value, by name, by need etc.
- The strategy used for binding the variables of a program: static binding (lexical scoping), according to the textual structure of the program, or dynamic binding (dynamic scoping), according to the dynamic chain of the program execution.
- The way recursive functions are written: textual recursion (the name of the function can be used as a free variable in the body of the function) or fixed-point recursion (the recursive function is nameless and is built using fixed-point combinators).

We name λ_0 our elementary, weak-typed, functional language that runs on the basic Lambda machine. In λ_0 types are not declared or automatically inferred. There are no type consistency checks while computing, and the Lambda machine never crashes. If some operation goes wrong, the result will be meaningless, although the machine will not stop with an error. In λ_0 values and operators are λ-expressions. A valid operation produces a λ-expression that represents the correct result. An invalid operation produces a λ-expression with no meaning. In λ_0 the freedom is total and the correctness of the computation relies solely on the programmer.

The following stages are used to define the language:

1. Implement values and operators for some basic data types (Booleans, pairs, lists, natural numbers). Show the way nameless recursive functions can be written and write simple programs using such functions.
2. Build specifications of the λ_0 language and discuss variants of binding the variables of a λ_0 program.
3. Modify the λ_0 specifications for accommodating textual recursion.

With these steps completed, some variants of the λ_0 language are close to real functional programming languages.

Initial conventions

We accept that parameter transfer in λ_0 is similar to the call by name (the reduction of λ-expressions is left-to-right or, equivalently, the normal-order evaluation is used). Therefore, the functions we will define are non-strict. Moreover, what is subject to evaluation is the function application only. Therefore, the functional normal form is taken as a normal form, i.e. we do not evaluate to body of a function prior to its application.

Whenever appropriate, the notation of λ-expressions with the format $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$ is abridged to $\lambda x_1\ x_2\ldots x_n.E$. Similarly, the application $(\ldots((\lambda x_1.\lambda x_2.\ \ldots\ \lambda x_n.E\ p_1)\ p_2)\ldots p_n)$ will be written $(\lambda x_1\ x_2\ldots x_n.E\ p_1\ p_2\ldots p_n)$. For example, the expression **λx.λy.(x y)** can be written more compactly as **λx y.(x y)**. The application **((λx.λy.(x y) a) b)** can be written either **(λx y.(x y) a b)** or, equivalently, **((λx y.(x y) a) b)**.

The notation ***symbol*** $\equiv_{\text{def}}$ λ-***expression*** defines the ***symbol*** as an identifier for the λ-***expression***. The ***symbol*** can be used as a free variable within other λ-expressions acting as a placeholder for the λ-***expression***. Whenever necessary the name will be textually substituted by its associated λ-***expression***. For sake of clarity, we may not substitute at once all the free occurrences of the ***symbol*** within the λ-***expression***, although "theoretically" we should. We forbid the use of the ***symbol*** as a free variable within the λ-***expression*** itself. This is in line with Lambda calculus where λ-expressions must be textually finite and where free variables have nothing to do with defining recursive functions.

The notation $E_1 = E_2$ is used to designate both the structural and the βη equivalence of E_1 and E_2. In particular, if the two expressions have normal form, $E_1 = E_2$ means that the result produced by E_1 is the same as the result produced by E_2 either by direct evaluation or by application on a number of identical parameters.

I. Defining values and operators for various data types

Many constructs from λ_0 use the following two functions, called **T** and **F**. These behave as selectors when applied on two arguments and, moreover, stand as an encoding of Boolean values.

T $\equiv_{\text{def}}$ **λx y.x**
F $\equiv_{\text{def}}$ **λx y.y**

$(T\ \ a\ \ b)\ \longrightarrow^*\ a$ and $(F\ \ a\ \ b)\ \longrightarrow^*\ b$

1. Boolean values and the conditional if...then...else

The values **true** and **false** are directly represented by the functions **T** and **F**. This encoding makes it easy to implement the basic Boolean operators.

true $\equiv_{\text{def}}$ **T** **or** $\equiv_{\text{def}}$ **λx y.(x T y)**
false $\equiv_{\text{def}}$ **F** **and** $\equiv_{\text{def}}$ **λx y.(x y F)**
not $\equiv_{\text{def}}$ **λx.(x F T)**

The truth tables for **not**, **and** and **or** can be easily verified. Since in λ_0 we are working with non-strict functions, **(or a b)** does not evaluate **b** if **a** is **true**, and **(and a b)** does not evaluate **b** if **a** is **false**. Therefore, **and** and **or** are similar to the **&&** and **||** C operators.

The conditional **if...then...else** is implemented as a ternary operator based on the selection properties of **true** and **false**.

if $\equiv_{\text{def}}$ λ**p a b.(p a b)**

(if true a b) $\longrightarrow^*$ **(T a b)** $\longrightarrow^*$ **a**

(if false a b) $\longrightarrow^*$ **(F a b)** $\longrightarrow^*$ **b**

Notice that the behavior of **if** is correct only if the parameters are transferred by name, as convened. It the reduction is left-to-right, the operands **a** and **b** are evaluated only when needed, after the evaluation of the predicate **p**.

2. Pairs

A pair corresponds to a value of the Cartesian product of two sets. Since λ_0 is weak-typed, the sets cover all possible λ-expressions. The pair **(a,b)** is represented as the function λ**z.(z a b)**. There are two projection operators for a pair: **fst** (abbreviation for first) returns the first element of a pair, and **snd** (abbreviation for second) returns the second element of a pair. The constructor of a pair takes two values and builds the function that represents the pair.

pair $\equiv_{\text{def}}$ λ**x y z.(z x y)**

fst $\equiv_{\text{def}}$ λ**p.(p T)**

snd $\equiv_{\text{def}}$ λ**p.(p F)**

It is verified that **(fst (pair a b))** $\longrightarrow^*$ **((pair a b) T)** $\longrightarrow^*$ **(T a b)** $\longrightarrow^*$ **a**

(snd (pair a b)) $\longrightarrow^*$ **((pair a b) F)** $\longrightarrow^*$ **(F a b)** $\longrightarrow^*$ **b**

3. Lists

A unidirectional list **L** is either:

- The empty list, which is called **nil**.
- A pair **P** such that the first element of **P** is the first element of **L** and the second element of **P** is a list (the tail of **L**).

The list constructor, called **cons**, is equivalent to **pair**. Similarly, **nil** is represented as a function that helps the predicate **null**, which tests if a list is empty, to behave correctly. In the spirit of LISP tradition, we name **car** the function that returns the first element of a non-empty list, and **cdr** the function that returns the tail of a non-empty list.

nil $\equiv_{\text{def}}$ λ**x.true** **---** the empty list

cons $\equiv_{\text{def}}$ λ**x y z.(z x y)** **---** similar to **pair**

car $\equiv_{\text{def}}$ λ**L.(L T)** **---** similar to **fst**

cdr $\equiv_{\text{def}}$ λ**L.(L F)** **---** similar to **snd**

null $\equiv_{\text{def}}$ λ**L.(L** λ**x y.false)**

We expect that **(car (cons a b))** $\longrightarrow^*$ **a**
(cdr (cons a b)) $\longrightarrow^*$ **b**
(null (cons a b)) $\longrightarrow^*$ **false**
(null nil) $\longrightarrow^*$ **true**

(cons a b) = (λx y z.(z x y) a b) $\longrightarrow^*$ **λz.(z a b)**

(car (cons a b)) =
(λL.(L T) λz.(z a b)) $\longrightarrow$ **(λz.(z a b) T)** $\longrightarrow$ **(T a b)** $\longrightarrow^*$ **a**

(cdr (cons a b)) =
(λL.(L F) λz.(z a b)) $\longrightarrow$ **(λz.(z a b) F)** $\longrightarrow$ **(F a b)** $\longrightarrow^*$ **b**

(null nil) =
(λL.(L λx y.false) λx.true) $\longrightarrow$ **(λx.true λx y.false)** $\longrightarrow$ **true**

(null (cons a b)) =
(λL.(L λx y.false) λz.(z a b)) $\longrightarrow$ **(λz.(z a b) λx y.false)**
$\longrightarrow$ **(λx y.false a b)** $\longrightarrow^*$ **false**

4. Natural numbers

There are various possible representations of numerals. Here we prefer a very simple variant: we consider a numeral as a list with as many elements as the value of the numeral. Zero corresponds to the empty list. As far as the elements are concerned, they are not important. We can choose the value **nil** for example. With this representation, it is easy to define two basic functions: **succ**, which computes the successor of a given numeral, and **pred**, which computes the predecessor of a strictly positive numeral.

0 $\equiv_{def}$ **nil**
succ $\equiv_{def}$ **λn.(cons nil n)**
pred $\equiv_{def}$ **λn.(n F)**   **---** actually the **cdr** list selector

Certainly, we need some more operators for lists and numerals just to be able to solve non-trivial problems in the λ_0 language. Operators such as list concatenation, integer addition, subtraction, and division can be defined recursively. For example, appending two generic lists **A** and **B** can be implemented as:

append $\equiv_{def}$ **λA B.(if (null A) B (cons (car A) (append (cdr A) B)))**

However, the above definition contradicts our convention, which forbids the use of the identifier of a λ-expression within the expression itself. Indeed, substituting **append** within its definition by its definition would produce a textually non-finite expression. For being able to define recursive functions we need an additional computational device: fixed-point combinators.

5. Fixed-point combinators and recursion

A fixed-point combinator is a function **F** such that for any function **f** there exists the equivalence: **(F f) = (f (F f))**.

Theorem 1. (the fixed-point theorem) $\forall \texttt{f}\in\Lambda\bullet(\exists \texttt{A}\in\Lambda\bullet\texttt{(f A) = A})$, i.e. for any λ-expression **f** there is a λ-expression **A** such that **(f A) = A**. The expression **A** is a fixed-point of **f**.

Proof. Let $\texttt{E} \equiv_{\text{def}} \lambda\texttt{x.(f (x x))}$ and $\texttt{A} \equiv_{\text{def}} \texttt{(E E)}$. Therefore, $\texttt{A} = (\lambda\texttt{x.(f (x x)) E)}$
$\longrightarrow \texttt{(f (E E)) = (f A)}$ ■

Corollary 1. $\exists\texttt{F}\in\Lambda\bullet(\forall\texttt{f}\in\Lambda\bullet\texttt{(F f) = (f (F f))})$, i.e. there exists a λ-expression **F** such that for any λ-expression **f** there is the equivalence **(F f) = (f (F f))**.

The corollary asserts the existence of fixed-point combinators. Choose $\texttt{F} \equiv_{\text{def}} \lambda\texttt{f.A}$, where **A** is the expression from the fixed-point theorem. We have $\texttt{(F f) = A}_{\texttt{[f/f]}}\texttt{ = A}$. Since **A = (f A)**, conformant to the fixed-point theorem, we get **(F f) = (f (F f))**.

Even without the help of the fixed-point theorem, if we take

$$\texttt{F} \equiv_{\text{def}} \lambda\texttt{f.(}\lambda\texttt{x.(f (x x)) }\lambda\texttt{x.(f (x x)))}^{1}$$

it can be verified that:

$\texttt{(F f) =}$
$(\lambda\texttt{f.(}\lambda\texttt{x.(f (x x)) }\lambda\texttt{x.(f (x x))) f)} \longrightarrow (\lambda\texttt{x.(f (x x)) }\lambda\texttt{x.(f (x x)))}$
$\qquad\qquad\qquad\qquad\qquad\qquad\longrightarrow \texttt{(f (}\lambda\texttt{x.(f (x x)) }\lambda\texttt{x.(f (x x))))}$
$\texttt{= (f (F f))}$ ■

The function $\texttt{F} \equiv_{\text{def}} \lambda\texttt{f.(}\lambda\texttt{x .(f (x x)) }\lambda\texttt{x .(f (x x)))}$ is not the only possible fixed-point combinator. For example, another fixed-point combinator is the expression **(C C)**, with $\texttt{C} \equiv_{\text{def}} \lambda\texttt{x.}\lambda\texttt{f.(f ((x x) f))}$. It can be verified that **(C C)** satisfies the property of a fixed-point combinator **((C C) f) = (f ((C C) f))**.

Fixed-point combinators can be used to build *anonymous* recursive functions. From mathematics we know that a function $\texttt{f:N} \to \texttt{N}$, where **N** is the set of natural numbers, is primitive recursive if $\texttt{f(0) = e}_{\texttt{0}}$ and **f(succ(n)) = g(n,f(n))**, where **g** is a function $\texttt{g:N}\times\texttt{N} \to \texttt{N}$. In this definition the names of functions are simply a commodity device. The property above can be expressed without explicitly labeling the functions.

Indeed, assume that the function **length**, which computes the length of a list, is defined as an anonymous function by using the fixed-point combinator **F** as follows:

$\texttt{lg} \equiv_{\text{def}} \lambda\texttt{cont L.(if (null L) 0 (succ (cont (cdr L))))}$
$\texttt{length} \equiv_{\text{def}} \texttt{(F lg) =}$
$\qquad\qquad \texttt{(lg (F lg))} \longrightarrow \lambda\texttt{L.(if (null L) 0 (succ (}\underbrace{\texttt{(F lg)}}_{\texttt{length}}\texttt{ (cdr L))))}$

$$\texttt{length} \equiv_{\text{def}} \lambda\texttt{L.(if (null L) 0 (succ (}\underbrace{\texttt{(F lg)}}_{\texttt{length}}\texttt{ (cdr L))))}$$

From the above definition of **length** it can be seen that the application **(F lg)** from within the body of the function has the same computational effect as **length** itself. The function

[1] The expression **F** is in functional normal form and, according to the conventions in $\lambda_{\mathbf{0}}$, it cannot be reduced further.

`λL.(if (null L) 0 (succ ((F lg) (cdr L))))` is recursive although its name is not used within its body. As far as the free variables **`F`**, **`if`**, **`succ`**, **`0`**, **`cdr`**, **`null`** and **`lg`** are concerned, they can be substituted by the λ-expressions they stand for. It is no point in doing here this substitution and contemplating an intricate expression.

Notice again that the construction works correctly due to the normal-order evaluation. Indeed, in the application **`(lg (F lg))`** the actual parameter **`(F lg)`** is not evaluated. It will be evaluated only when necessary, during the evaluation of the body of the function **`lg`**. When happens, the evaluation **`(F lg)`** replicates the behavior of the computation that is conventionally labeled **`length`**.

What will happen with the definition of **`length`** if its corresponding λ-expression is subject to the applicative-order evaluation (i.e. transfer by value)? The evaluation of **`(lg (F lg))`** does not terminate! Can we find a way out and have anonymous recursive functions even if the parameters are transferred by value? The functions c_1 and c_2 below are fixed-point combinators and they help defining anonymous unary and binary recursive functions that work correctly even it they evaluate their parameters when applied.

`c1` $\equiv_{def}$ **`λf.(λg x.(f (g g) x)  λg x.(f (g g) x))`**
`c2` $\equiv_{def}$ **`λf.(λg x y.(f (g g) x y)  λg x y.(f (g g) x y))`**

Theorem 4. The functions **`c1`** and **`c2`** are fixed-point combinators.

Proof for **`c1`**. We have to show that **`(c1 Fun) = (Fun (c1 Fun))`**, for any function **`Fun`**.

$$
\begin{aligned}
\texttt{(c1 Fun)} &= \texttt{(}\lambda\texttt{f.(}\lambda\texttt{g x.(f (g g) x)}\ \ \lambda\texttt{g x.(f (g g) x)) Fun)}\\
&\longrightarrow \underbrace{\texttt{(}\lambda\texttt{g x.(Fun (g g) x)}\ \ \lambda\texttt{g x.(Fun (g g) x))}}_{\texttt{(c1 Fun)}}\\
&\longrightarrow \lambda\texttt{x.(Fun (}\lambda\texttt{g x.(Fun (g g) x)}\ \ \lambda\texttt{g x.(Fun (g g) x))}\ \ \texttt{x)}\\
&= \lambda\texttt{x.(Fun (c1 Fun) x)}
\end{aligned}
$$

Notice that **`λx.(Fun (c1 Fun) x)`** is not evaluated since it is in functional normal form. In addition, **`((c1 Fun) a) = (λx.(Fun (c1 Fun) x) a)`** $\longrightarrow$ **`(Fun (c1 Fun) a)`**. But **`(Fun (c1 Fun) a)`** is the abbreviation of **`((Fun (c1 Fun)) a)`**, and since **`a`** is an arbitrary value we conclude that **`(c1 Fun) = (Fun (c1 Fun))`** ■

The proof for **`c2`** is similar. Observe the particularity of these combinators. The equation **`(c1 Fun) = λx.(Fun (c1 Fun) x)`** shows that: a) the evaluation of **`(c1 Fun)`** terminates with a functional normal form and b) when *applied* on a parameter **`a`** the function **`(c1 Fun)`** has the same computational effect as **`(Fun (c1 Fun) a)`**. This explains the definition of anonymous recursive functions that suit the applicative-order evaluation. Take again the property of **`c1`**:

$$\texttt{((c1 Fun) a)} \longrightarrow^{*} \texttt{(Fun (c1 Fun) a)}$$

and consider the function **`Fun`** as

$$\texttt{Fun} \equiv_{def} \lambda\texttt{cont x.(if (p x) }e_0\texttt{ (G x (cont (Q x))))}$$

Also define the (recursive) function **`Fr`** $\equiv_{def}$ **`(c1 Fun)`** = **`λx.(Fun cont x)`**, where the free variable **`cont`** stands for the expression **`(c1 Fun)`**, therefore for **`Fr`** itself.

$$
\begin{aligned}
\texttt{(Fr a)} &\longrightarrow \texttt{(Fun cont a)}\\
&\longrightarrow^{*} \texttt{(if (p a) }e_0\texttt{ (G a (cont (Q a))))}
\end{aligned}
$$

Case **(p a) = true**.

(Fr a) = (if (p a) e$_0$ (G a (cont (Q a)))) $\longrightarrow^*$ **e$_0$**

Case **(p a) = false**.

(Fr a) = (if (p a) e$_0$ (G a (cont (Q a))))

$\longrightarrow^*$ **(G a (cont (Q a))) = (G a ((c1 Fun) (Q a))) = (G a (Fr (Q a)))**

Therefore **Fr** is a recursive function. In a similar way it can be verified that **Fr'** below is a (curried) binary recursive function

Fr'$\equiv_{def}$ **(c2 Fun)**, where
Fun $\equiv_{def}$ **λcont x y.(if (p x y) e$_0$ (G x y (cont (Q x y) (R x y))))**,

(Fr' a b) = e$_0$ if **(p a b) = true**
(Fr' a b) = (G a b (Fr′ (Q a b) (R a b))), if **(p a b) = false**.

6. Recursive operators for lists and numbers

Using the fixed-point combinators **c1** and **c2** we can implement recursive functions for lists and numbers in the λ$_0$ language. Below are few of them. They will work just fine in real functional programming languages.

```
append ≡def (c2 λcont A B.(if (null A) B
                              (cons (car A) (cont (cdr A) B))))
```

(append A B) returns the concatenation of the lists **A** and **B**. In another variant, observing that the second parameter of **append** does not change, we can use **c1** instead of **c2**.

```
append ≡def  λA B.((c1 λcont A.(if (null A) B
                                   (cons (car A)(cont (cdr A)))) A)
```

```
length ≡def (c1  λ cont L.(if (null L) 0 (succ (cont (cdr L)))))
```
Reversing a list **L** can be achieved by **(reverse L)** defined as:

```
reverse ≡def λL.((c2 rv) nil L)
rv ≡def λ cont R L.(if (null L) R (cont (cons (car L) R) (cdr L)))
```

For conveniently defining operations with natural numbers we need few comparison operators. Here **(zerop n)** returns **true** if **n** is zero and **false** otherwise; **(lt n m)** returns **true** if **n** is smaller than **m**, whereas **(eq n m)** returns **true** if **n** and **m** are equal.

```
zerop ≡def null
lt ≡def (c2 λcont n m.(if (zerop m) false
                          (if (zerop n) true (cont (pred n) (pred m)))))

eq ≡def (c2 λcont n m.(if (zerop n) (zerop m)
                          (if (zerop m) false (cont (pred n) (pred m)))))
```

The addition of two natural numbers is immediate. Indeed, since the two numbers are represented as lists, each of equal length to the value of a number, the result of addition is the list obtained by appending the two lists. Subtraction is defined recursively.

```
+ ≡def append
- ≡def (c2 λcont x y.(if (zerop y) x (cont (pred x) (pred y))))
```

The integer multiplication **`*`**, the integer division **`div`**, and the modulo operation **`mod`** are defined recursively as:

```
* ≡def λx y.((c1  λcont x.(if (zerop x) 0  (+ y (cont (pred x))))) x)
div ≡def λx y.((c1  λcont x.(if (lt x y) 0  (succ (cont (– x  y))))) x)
mod ≡def λx y.((c1  λcont x.(if (lt x y) x  (cont (– x  y)))) x)
```

The functions above work properly in real functional programming languages. Below is given the Caml transcription of the function **`length`**.

```
let rec c1 = fun Fun x -> Fun (c1 Fun) x;;[2]

let length = c1 (fun cont L -> if L=[] then 0
                                       else (succ (cont (tl L))));;
```

7. Infinite lists. Streams

Using normal-order evaluation in the λ_0 language makes it possible to work with infinite objects such as possible infinite lists, called also *streams* in the slang of functional programming community. The fact that while constructing a list cell by **`(cons a b)`** the parameter **`b`** is not evaluated allows **`b`** to be a sort of computing device that stepwise unfolds an eventually infinite list tail.

Consider as an example the infinite list, called **`natural`**, of natural numbers and the selector of the nth number from the list.

```
natural ≡def ((c1 λcont k.(cons k (cont (succ k)))) 0)
nth  ≡def (c2 λcont L n.(if (zerop n) (car L) (cont (cdr L) (pred n))))
```

(nth natural 0) = (car natural) $\longrightarrow^*$ 0

(nth natural n) = (car (cdr^n natural)) $\longrightarrow^*$ n, for n > 0

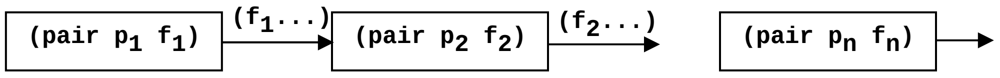

Figure 1. A stream representation that suits the applicative-order evaluation

Notice that the applicative-order evaluation of the expression **`(cons k (cont (succ k)))`** does not terminate. Indeed, the result is the infinite list of natural numbers. However,

[2] For simplicity, the Caml definition of the fixed-point combinator **`c1`** is textually recursive. It can be written in a textually non-recursive format as shown in a subsequent lecture.

if the stream is represented in a finite way the termination problem is solved regardless the parameter transfer mode that is used. For instance, the stream could be represented as a pair **(pair p_1 f_1)**, where p_1 is the first term of the stream and f_1 is a function able to compute - when called - the next finite part **(pair p_2 f_2)** of the stream, as illustrated in figure 1. The stream unfolds in a controlled fashion, only when needed. The details of this kind of stream representation are discussed in a subsequent lecture.

8. Writing λ_0 programs

Even with the scant data and operators we already have, λ_0 programs can be written much like in any real functional programming language.

Consider writing a function, called **primes**, that computes the stream of all prime numbers. The **primes** function uses the auxiliary **is_prime** predicate: **(is_prime n)** tests if **n** is prime. Although this is not the most efficient solution, it is accepted here for the sake of simplicity.

```
is_prime ≡def λn.((c1 λcont divisor.(if (eq divisor 1)
                                        true
                                        (and (not (zerop (mod n divisor)))
                                             (cont (pred divisor)))))
                 (div n 2))

primes ≡def ((c1 λcont n.(if (is_prime n) (cons n (cont (succ n)))
                                          (cont (succ n)))) 2)
1 ≡def (succ 0)
2 ≡def (succ 1)
```

The program written in λ_0 can be encoded with minor modifications in Haskell, a functional programming language with normal-order evaluation. Apart from minor syntactic aspects, the differences are motivated by the strong typing in Haskell (that automatically infers the types of the program values and variables). The type system of Haskell prohibits the definition of the **c1** combinator following strictly the pattern used in λ_0. For simplicity, we adopt a textual recursive definition of **c1**. The program contains the additional function **nth**. The application **(nth n *stream*)**, $n \geq 0$, returns the nth element of the (possible infinite) ***stream*** of values.

```
-- Haskell code for the stream of prime numbers
module Primes where
c1 f= f (c1 f)
is_prime n =
   (c1 (\ rec d -> if d==1 then True else (n`mod`d /= 0)&&(rec(d-1))))
   (n `div` 2)

primes = (c1 (\ rec n -> if is_prime n then n:rec(n+1) else rec(n+1))) 2
nth n from = head (drop (n-1) from)
-- end of module

nth 81 primes
419 :: Integer
```