Evaluating untyped Lambda expressions in Haskell

Supervisor: prof. Alessi Fabio
Co-Superisor: prof. Di Gianantonio Pietro

Author: D’Ambrosi Denis

Scuola Superiore «Di Toppo Wassermann», October 2023

Table of Contents

1. Pure λ-Calculus

2 key ingredients

A set of λ-terms Λ

x is a variable $\rightarrow x \in \Lambda$
x is a variable, $M \in \Lambda \rightarrow (\lambda x. M) \in \Lambda$
$M, N \in \Lambda \rightarrow (M\ N) \in \Lambda$

+

Three reduction rules

α–conversion: $(\lambda x. M) \mapsto (\lambda y. M[x \coloneqq y])$
β–conversion: $\big((\lambda x. M)\ N\big) \mapsto M[x \coloneqq N]$
η–conversion: $\big((\lambda x. M)\ x\big) \mapsto M$ if $x \notin FV(M)$

… and a recursive formula

$FV(x) = x$ if x is a variable
$FV\big((\lambda x. M)\big) = FV(M) \backslash \mathrm{x}$
$FV\big((M\ N)\big) = FV(M) \cup FV(N)$

1. Pure λ-Calculus

Why is it still relevant?

- Symple syntax and semantics, but still **Turing complete**!
- Widely implemented in **modern programming** languages

Exploring the Power of JavaScript Lambda Expressions

André Ericson · September 11, 2023

The Journal of Symbolic Logic
Volume 2, Number 4, December 1937

COMPUTABILITY AND λ-DEFINABILITY

A. M. TURING

Several definitions have been given to express an exact meaning corresponding to the intuitive idea of 'effective calculability' as applied for instance to functions of positive integers. The purpose of the present paper is to show that the computable[1] functions introduced by the author are identical with the λ-definable[2] functions of Church and the general recursive[3] functions due to Herbrand and Gödel and developed by Kleene. It is shown that every λ-definable function is computable and that every computable function is general recursive. There is a modified form of λ-definability, known as λ-*K*-definability, and it turns out to be natural to put the proof that every λ-definable function is computable in the form of a proof that every λ-*K*-definable function is computable; that every λ-definable function is λ-*K*-definable is trivial. If these results are taken in conjunction with an already available[4] proof that every general recursive function is λ-definable we shall have the required equivalence of computability with λ-definability and incidentally a new proof of the equivalence of λ-definability and λ-*K*-definability.

A definition of what is meant by a computable function cannot be given satisfactorily in a short space. I therefore refer the reader to *Computable* pp. 230–235 and p. 254. The proof that computability implies recursiveness requires no more knowledge of computable functions than the ideas underlying the definition: the technical details are recalled in §5. On the other hand in proving that the λ-*K*-definable functions are computable it is assumed that the reader is familiar with the methods of *Computable* pp. 235–239.

The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions. For those who take this view the formal proof of equivalence provides a justification for Church's calculus, and allows the 'machines' which generate computable functions to be replaced by the more convenient λ-definitions.

Received September 11, 1937.

[1] A. M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, *Proceedings of the London Mathematical Society*, ser. 2, vol. 42 (1936–7), pp. 230–265, quoted here as *Computable*. A similar definition was given by E. L. Post, *Finite combinatory processes—formulation 1*, this Journal, vol. 1 (1936), pp. 103–105.

[2] Alonzo Church, *An unsolvable problem of elementary number theory*, *American journal of mathematics*, vol. 58 (1936), pp. 345–363, quoted here as *Unsolvable*.

[3] S. C. Kleene, *General recursive functions of natural numbers*, *Mathematische Annalen*, vol. 112 (1935–6), pp. 727–742. A definition of general recursiveness is also to be found in *Unsolvable* pp. 350–351.

[4] S. C. Kleene, *λ-definability and recursiveness*, *Duke mathematical journal*, vol. 2 (1936), pp. 340–353.

153

Harnessing the Elegance of Python's Lambda Functions

Discover the power of one-liner functions in Python

Patrick Kalkman · Follow
Published in ITNEXT · 11 min read · May 16

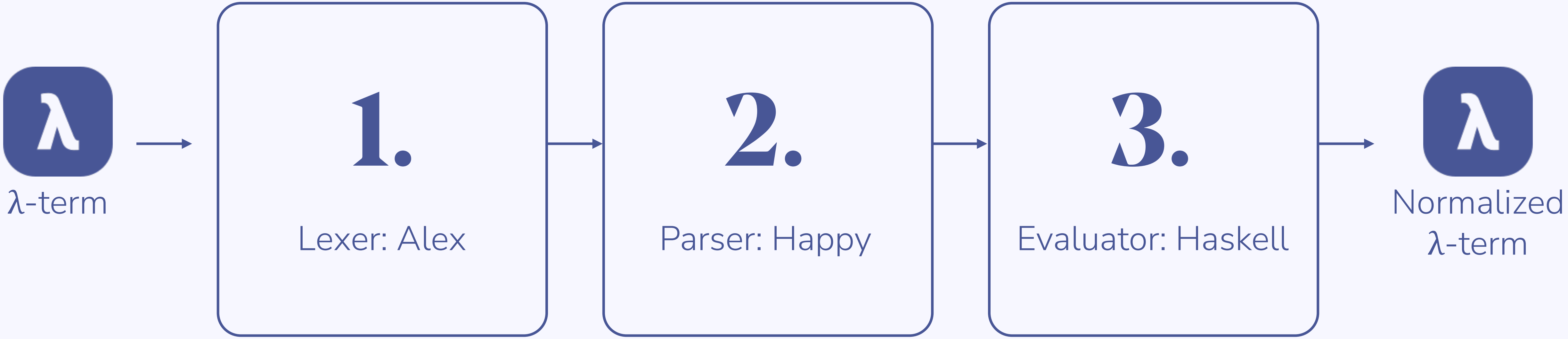
109 1

Java Lambda Expressions: A Comprehensive Guide 101

Manisha Jena • Last Modified: December 29th, 2022

2. Evaluating λ-expressions

Our pipeline:



2.1 Lexing λ-expressions

Custom data type for tokens

```
data Token = NewLine
           | Term String
           | Dot
           | Lambda
           | OpenPar
           | ClosedPar
deriving (Eq, Show)
```

+

Regexes for recognising the lexemes

```
$letter = [a-zA-Z]

tokens :-
       \n            { \s → NewLine }
       $letter+      { \s → Term s }
       \.            { \s → Dot }
       \λ            { \s → Lambda }
       \(            { \s → OpenPar }
       \)            { \s → ClosedPar }
       $white        ;
```

Note that the lexer only recognises alphabetical identifiers!

2.2 Parsing λ-expressions

Custom data type for λ-terms

```
data Term = Var String
          | Abs String Term
          | App Term Term
          | Empty
deriving (Eq)
```

\+

Grammar specification with actions

```
%left ‘λ’ ‘.’ VAR

start  : line                  { [$1] }
       | start line            { $2 : $1 }

line   : NL                    { Empty }
       | exp NL                { $1 }

exp    : VAR                   { Var $1 }
       | ‘(‘ exp ‘)’           { $2 }
       | ‘λ’ VAR ‘.’ exp       { Abs $2 $4 }
       | exp exp               { App $1 $2 }
```

A set of λ-terms Λ

x is a variable → $x \in \Lambda$
x is a variable, $M \in \Lambda \rightarrow (\lambda x. M) \in \Lambda$
$M, N \in \Lambda \rightarrow (M\ N) \in \Lambda$

2.3 Evaluating λ-expressions: the algebraic approach

Implementation of β-conversion
$$((\lambda x. M)\ N) \mapsto M[x \coloneqq N]$$

+

A strategy to handle name clashes

α-conversion

or

De Bruijn Indexes

2.3.1 α-conversion (eager)

```
eval :: Term → Term
eval (App (Abs x t) t1) = let t1’ = eval t1 in eval (sub x t t1’)
eval (App t1 t2) = case eval t1 of
      a@(Abs x t1) → eval (App a t2)
      t → App t (eval t2)
eval (Abs x t) = Abs x (eval t)
eval Empty = error «Empty detected»
eval t = t
```

A form of β-reduction with strict semantics is performed in the first pattern

2.3.1 α-conversion (lazy)

```
eval :: Term → Term
eval (App (Abs x t) t1) = eval (sub x t t1)
eval (App t1 t2) = case eval t1 of
        a@(Abs x t1) → eval (App a t2)
        t → App t (eval t2)
eval (Abs x t) = Abs x (eval t)
eval Empty = error «Empty detected»
eval t = t
```

A form of β-reduction with lazy semantics is performed in the first pattern

2.3.1 α-conversion

Don’t substitute further to allow name shadowing

```haskell
sub :: String -> Term -> Term -> Term
sub x v@(Var y) newVal
      | x == y = newVal
      | otherwise = v
sub x a@(Abs y t) newVal
      | x == y = a
      | notElem y (freeVars newVal) = Abs y (sub x t newVal)
      | otherwise = let y’ = firstNameAvailable (freeVars newVal) in
            Abs y’ (sub x (alphaConv y t y’) newVal)
sub x (App t1 t2) t = App (sub x t1 t) (sub x t2 t)
sub _ Empty _ = error "Empty detected"
```

2.3.1 α-conversion

If the variable does not appear free, we can substitute

```
sub :: String -> Term -> Term -> Term
sub x v@(Var y) newVal
      | x == y = newVal
      | otherwise = v
sub x a@(Abs y t) newVal
      | x == y = a
      | notElem y (freeVars newVal) = Abs y (sub x t newVal)
      | otherwise = let y' = firstNameAvailable (freeVars newVal) in
            Abs y' (sub x (alphaConv y t y') newVal)
sub x (App t1 t2) t = App (sub x t1 t) (sub x t2 t)
sub _ Empty _ = error "Empty detected"
```

2.3.1 α-conversion

Otherwise, we first need to apply α-conversion

```
sub :: String → Term → Term → Term
sub x v@(Var y) newVal
      | x == y = newVal
      | otherwise = v
sub x a@(Abs y t) newVal
      | x == y = a
      | notElem y (freeVars newVal) = Abs y (sub x t newVal)
      | otherwise = let y' = firstNameAvailable (freeVars newVal) in
            Abs y' (sub x (alphaConv y t y') newVal)
sub x (App t1 t2) t = App (sub x t1 t) (sub x t2 t)
sub _ Empty _ = error "Empty detected"
```

2.3.1 α-conversion

α-conversion is nothing more than a simple renaming of variables and bindings

```haskell
alphaConv :: String -> Term -> String -> Term
alphaConv x v@(Var y) z
       | x == y = Var z
       | otherwise = v
alphaConv x (Abs y t) z
       | x == y = Abs z (alphaConv x t z)
       | otherwise = Abs y (alphaConv x t z)
alphaConv x (App t1 t2) z = App (alphaConv x t1 z) (alphaConv x t2 z)
alphaConv _ Empty _ = error "Empty detected"
```

2.3.1 α-conversion

Additional notes

Names

Unlimited names
for α-converion
(lazy evaluation)

```
variableSet :: [String]
variableSet = concatMap (\k → replicateM k ['a'..'z']) [1..]
```

2.3.1 α-conversion

Additional notes

Names

Unlimited names
for α-converion
(lazy evaluation)

FV

`freeVars`
derived from the
definition of FV

```
freeVars :: Term → [String]
freeVars (Var x) = [x]
freeVars (Abs x t1) = (freeVars t1) \\ [x]
freeVars (App t1 t2) = (freeVars t1) ++ (freeVars t2)
```

2.3.2 De Bruijn Indexes

De Bruijn notation substitutes variables symbols with the **distance from their bindings**

$$(\lambda x.\, x) \mapsto (\lambda\ 1)$$
$$(\lambda x.\, c) \mapsto (\lambda\ c)$$
$$\left(\lambda x.\, \left(\lambda y.\, (\lambda z.\, (x\ z)\ c)\right)\right) \mapsto \left(\lambda \left(\lambda \left(\lambda((3\ 1)\ c)\right)\right)\right)$$

This effectively removes variables (and name clashes as a result)

2.3.2 De Bruijn Indexes

De Bruijn notation substitutes variables symbols with the **distance from their bindings**

$$(\lambda x.\, x) \mapsto (\lambda\ 1)$$
$$(\lambda x.\, c) \mapsto (\lambda\ c)$$
$$\left(\lambda x.\, \left(\lambda y.\, (\lambda z.\, (x\ z)\ c)\right)\right) \mapsto \left(\lambda \left(\lambda \left(\lambda((3\ 1)\ c)\right)\right)\right)$$

This effectively removes variables (and name clashes as a result)

We will need a new data type to handle this new form:

```
data DBTerm = DBVar Int
            | DBConst String
            | DBAbs DBTerm
            | DBApp DBTerm DBTerm
deriving (Eq)
```

2.3.2 De Bruijn Indexes

```
toDeBruijn :: Term → DBTerm
toDeBruijn = toDBRec [] where
      toDBRec :: [String] → Term → DBTerm
      toDBRec vars (Var x) = case elemIndex x vars of
            Just n → DBVar (n+1)
            Nothing → DBConst x
      toDBRec vars (Abs x t) = DBAbs (toDBRec (x:vars) t)
      toDBRec vars (App t1 t2) = DBApp (toDBRec vars t1)  (toDBRec vars t2)
      toDBRec _ Empty = error "Empty detected"
```

We exploit the `Maybe` monad to discriminate between variables and constants

2.3.2 De Bruijn Indexes

```
toDeBruijn :: Term → DBTerm
toDeBruijn = toDBRec [] where
      toDBRec :: [String] → Term → DBTerm
      toDBRec vars (Var x) = case elemIndex x vars of
            Just n → DBVar (n+1)
            Nothing → DBConst x
      toDBRec vars (Abs x t) = DBAbs (toDBRec (x:vars) t)
      toDBRec vars (App t1 t2) = DBApp (toDBRec vars t1)  (toDBRec vars t2)
      toDBRec _ Empty = error "Empty detected"
```

By managing the set of bindings as a stack, we allow variable shadowing

2.3.2 De Bruijn Indexes (eager)

```
eval :: Term -> DBTerm
eval t = dbEval (toDeBruijn t) where
      dbEval :: DBTerm -> DBTerm
      dbEval (DBApp (DBAbs t) t1) = let t1' = dbEval t1 in dbEval (sub t t1' 1)
      dbEval (DBApp t1 t2) = case dbEval t1 of
            a@(DBAbs t) -> dbEval (DBApp a t2)
            t -> DBApp t (dbEval t2)
      dbEval (DBAbs t) = DBAbs (dbEval t)
      dbEval t = t
```

A form of β-reduction with strict semantics is performed in the first pattern

2.3.2 De Bruijn Indexes (lazy)

```
eval :: Term -> DBTerm
eval t = dbEval (toDeBruijn t) where
      dbEval :: DBTerm -> DBTerm
      dbEval (DBApp (DBAbs t) t1) = dbEval (sub t t1 1)
      dbEval (DBApp t1 t2) = case dbEval t1 of
            a@(DBAbs t) -> dbEval (DBApp a t2)
            t -> DBApp t (dbEval t2)
      dbEval (DBAbs t) = DBAbs (dbEval t)
      dbEval t = t
```

A form of β-reduction with lazy semantics is performed in the first pattern

2.3.2 De Bruijn Indexes

```haskell
sub :: DBTerm -> DBTerm -> Int -> DBTerm
sub (DBVar n) t n'
       | n == n' = t
       | otherwise = DBVar n
sub c@(DBConst _) _ _ = c
sub (DBAbs t1) t2 n = DBAbs (sub t1 t2 (n+1))
sub (DBApp t1 t2) t n = DBApp (sub t1 t n) (sub t2 t n)
```

Substitution is now straightforward: we only need to keep track of the level of nesting

2.4 Computational approach

We will need a new data type to exploit Haskell's anonymous functions in a type-safe way

```
data CompTerm = CompVar String
              | CompAbs (CompTerm → CompTerm)
              | CompApp CompTerm CompTerm
```

2.4 Computational approach

We will need a new data type to exploit Haskell's anonymous functions in a type-safe way

```
data CompTerm = CompVar String
       | CompAbs (CompTerm -> CompTerm)
       | CompApp CompTerm CompTerm
```

Note that including anonymous functions in the constructors prevents us from deriving **Eq** and **Show**

2.4 Computational approach

```
eval :: Term -> CompTerm
eval = evalRec [] [] where
    evalRec :: [CompTerm] -> [String] -> Term -> CompTerm
    evalRec env vars (Abs v t) = CompAbs (\ x -> evalRec (x:env) (v:vars) t)
    evalRec env vars (Var x) = case elemIndex x vars of
        Just n -> env !! n
        Nothing -> CompVar x
    evalRec env vars (App t1 t2) = let t2' = evalRec env vars t2 in
         case evalRec env vars t1 of
             CompAbs f -> f t2'
             v@(CompVar _) -> CompApp v t2'
             a@(CompApp _ _) -> CompApp a t2'
    evalRec _ _ Empty = error "Empty detected"
```

We have to build our own closures for the expressions

2.4 Computational approach

```haskell
eval :: Term -> CompTerm
eval = evalRec [] [] where
    evalRec :: [CompTerm] -> [String] -> Term -> CompTerm
    evalRec env vars (Abs v t) = CompAbs (\ x -> evalRec (x:env) (v:vars) t)
    evalRec env vars (Var x) = case elemIndex x vars of
        Just n -> env !! n
        Nothing -> CompVar x
    evalRec env vars (App t1 t2) = let t2' = evalRec env vars t2 in
         case evalRec env vars t1 of
              CompAbs f -> f t2'
              v@(CompVar _) -> CompApp v t2'
              a@(CompApp _ _) -> CompApp a t2'
    evalRec _ _ Empty = error "Empty detected"
```

When we reach the base case for the induction, we refer to the recursively built environment

2.4 Computational approach

```
eval :: Term → CompTerm
eval = evalRec [] [] where
    evalRec :: [CompTerm] → [String] → Term → CompTerm
    evalRec env vars (Abs v t) = CompAbs (\ x → evalRec (x:env) (v:vars) t)
    evalRec env vars (Var x) = case elemIndex x vars of
        Just n → env !! n
        Nothing → CompVar x
    evalRec env vars (App t1 t2) = let t2' = evalRec env vars t2 in
         case evalRec env vars t1 of
             CompAbs f → f t2'
             v@(CompVar _) → CompApp v t2'
             a@(CompApp _ _) → CompApp a t2'
    evalRec _ _ Empty = error "Empty detected"
```

Application is straighforward: just exploit Haskell's own anonymous function mechanisms

2.4 Computational approach

We have to find a way to print the evaluated expression, but we cannot print anonymous functions directly

➔ We can infere the structure of an abstraction by passing a parameter to it, and observing the result

For example, if we have an abstraction f and we knew that $(f\ a) = a$, then it must be true that f can be expressed as either $(\lambda x. x)$ or $(\lambda x. a)$

How can we discriminate betwen these two cases? By providing arguments that are surely not part of the original expressions: numerical strings!

Remember that our lexer only recognises alphabetical identifiers

2.4 Computational approach

Our printing procedure:

Black-box abstraction → **1.** Integers as arguments → **2.** Convert to `Term` → **3.** Substitute integers → Printable λ-term

3. Conclusions

Pros and cons of each approach

α-conversion

✓ Does not need translations
✓ «Standard» approach

✗ Have to implement α-conv

3. Conclusions

Pros and cons of each approach

α-conversion

✓ Does not need translations
✓ «Standard» approach

✗ Have to implement α-conv

De Bruijn Indexes

✓ Effortless substitution

✗ Need a translation
✗ Readability

3. Conclusions

Pros and cons of each approach

α-conversion

✓ Does not need translations
✓ «Standard» approach

✗ Have to implement α-conv

De Bruijn Indexes

✓ Effortless substitution

✗ Need a translation
✗ Readability

Computational

✓ Elegant applications
✓ No name clashes by default

✗ Need to create closures
✗ No default printing method