Evaluating untyped λ-expressions in Haskell

D'Ambrosi Denis
dambrosi.denis@spes.uniud.it
147681

October 3, 2023

Abstract

Contents

Introduction

In this report, we will explore various approaches to normalize expressions belonging to Church's λ-Calculus. In Section 1, we we briefly introduce the syntax and semantics of λ-Calculus, along with its reduction techniques. In Section 2 we will actually present the implementations of three different strategies to solve this problem. Finally, in Section 3 we will analyze the pros and cons of each approach and present some potential future work.

1 Pure λ-Calculus

Pure λ-Calculus, developed by Alonzo Church in the 1930s, is a formal system that plays a pivotal role in the foundation of theoretical computer science. It is a minimalist, yet immensely powerful, mathematical framework for expressing computation using only anonymous functions.

The set of λ-terms Λ is the minimal set that can be constructed inductively according the following rules (that effectively define the syntax of this formalism):

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M)$
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$

Rule number 2 expresses the possibility of creating *abstractions*. An abstraction is an anonymous function that takes as input a λ-term (in this case x) and returns a λ-term (not necessarily a variable, in this case M) as output. Multiple inputs functions can be constructed via a process called *currying*: since higher-order-functions are allowed within λ-Calculus, we can transform any n-ary function into an abstraction that takes only one input and returns another abstraction as output. The latter will, in turn, consume only one input and produce yet another abstraction, and so on, n times. Evaluating the last function will, eventually, produce the desired output.

Rule number 3, on the other hand, allows for the application of abstractions to λ-terms: performing such operation will yield a new λ-term according two main reduction operations:

1. *α-conversion* aims at renaiming bound variables for avoiding name collisions: $(\lambda x.M) \mapsto (\lambda y.M[x := y])$
2. *β-reduction* actually normalizes λ-terms by replacing bound variables in the body of the abstraction with the actual parameter: $((\lambda x.M)N) \mapsto M[x := N]$

Before executing a β-reduction step, we want to ensure that the actual parameter substituted within an abstraction's body does not appear *free* in it. Formally, provided a λ-term t, the set of its *Free variables* $FV(t)$ can be computed recursively as follows:

- $FV(x) = x$ if x is a variable
- $FV((\lambda x.M)) = FV(M) \setminus x$
- $FV((M\ N)) = FV(M) \cup FV(N)$

If the input of the application appears free within the abstraction, then we must execute an process of α-conversion on the latter to avoid any undesired change of semantics due to name clashing. If we did not take this additional care while substituting, we may inadvertly compute incorrect terms while β-reducing.

This quick overview of the syntax and semantics of the λ-Calculus has laid the groundwork for introducing the actual implementations of λ-expressions evaluators of the upcoming sections. Before continuing, it is worth discussing briefly the reason why is formalism is still relevant, even after all these years. In 1936, Church proved that λ-Calculus is Turing complete, meaning that it has the computational power equivalent to a universal Turing machine. This critical property establishes λ-Calculus as a foundational formal system in the study of computation: this results implies that any computable function can be expressed and evaluated within the confines of λ-Calculus. Such fact underpins its applications not only in theoretical computer science but also in practical domains, where it serves as the basis for functional programming languages and aids in the development of algorithms for various computational tasks.

2 Evaluating λ-expressions

> *A monad is a monoid in the cathegory of endofunctors*
>
> *Unknown*

Our objective consists in implementing an evaluator for λ-expressions that computes the normal form of each given λ-term. To provide a user-friendly-sh interface to our software, we allow users to write the input λ-expressions into a text file, that gets lexed and parsed before applying the reduction strategies. Our technology stack is completely Haskell-based and is structured as follows:

1. Alex as lexer
2. Happy as parser
3. The evaluation strategy is implemented in pure Haskell with some support from the Prelude ad List packages

In the following sections, we will give a brief overview of each phase by commenting the key pieces of code needed to perform the relative computation. Please note that the actual evaluation strategies are explained in detail in sections 2.3 and 2.4

2.1 Lexing λ-expressions

As previously introduced, our lexer generator of choice is Alex. Alex allows developers to define a set of regular expression-based rules to recognise lexems from a text stream and produce the relative tokens according to predetermined actions. In practice, we first need to declare the set of our syntactic tokens by defining an abstract data type with a constructor for each single token. In our case, we need to recognise variables (that can be modelled as alphabetical strings), the λ symbol, the . symbol, round parentheses, whitespaces and newlines characters (to allow the user to specify multiple expressions). In practice, this straightforwardly translates in the following Haskell datatype:

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

Now we can actually recognise lexemes through regular expressions. Alex's syntax allows user to associate names to regexes before use to improve readability: in our case, it will be helpful to define the language of single letters (both lower and uppercase):

```
$letter = [a–zA–Z]
```

We can then use the above expression in the lexemes-recognition rules:

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

Note that:

1. We have associated an action to each lexeme in the form of a (ironically) Haskell λ-expression that maps the read string into the relative token. In our case the actions were trivially defined since we are dealing with a very simple language, but there is not limit to the complexity of these functions. Supporting functions can be also defined along with the custom datatype.

2. The lexems are ordered according to descending priority: even though the newline character will be recognised by both the first and the last expression, only the first one will be applied to produce a NewLine token.

By compiling the file through Alex, we obtain a custom Haskell module that can be called from other modules to read any text stream and recognise its lexems. The following parser module will exploit such functionality to convert the input file into a stream of tokens that can be then converted into an abstract syntax tree.

2.2 Parsing λ-expressions

Following a very similar computational pipeline to compilers' frontends, the lexed tokens must be analyzed by a parser. In our case, we adopted the Happy parser generator, which, given a grammar specified in Backus-Naur Form, produces the Haskell code to parse it. The modules produced through Happy adopt a *LARL(1) bottom-up strategy* which is less powerful than the more general *LR(1)* approach, yet can be implemented in a more efficient way. Since diving more deeply into the expressiveness and computational complexity of parsing techniques is out of the scope of this report, the reader can be satisfied by knowing that:

1. Happy-generated parsers are expressive enough to recognise λ-Calculus terms (as a matter of fact, they are enough expressive to work in the GHC frontend!)

2. In the only case of this project where we would have benefited from a top-down parsing approach see the closure issue in Section 2.4, we easily overcame the problem by firstly building a syntax tree in a bottom-up fashion and then traversing the structure with a top-down algorithm.

The objective of our parser will be to recognise the syntactic structure of each λ-expression and to feed it as a reasonable data structure to the evaluator. A trivial data type suitable for the task can be defined as follows:

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

This abstract data type straightforwardly mirrors the construction of the set of λ-terms Λ introduced in Section 1. The Empty constructor is declared only to streamline the handling of empty lines in the input file and will not be considerated in the following evaluating strategies. As we can see, Term derives the standard Eq type class, but not Show's one. Instead, we decided to implement a custom function to pretty-print functions:

```
instance Show Term where
    show :: Term -> String
    show (Var x) = x
```

```
show (Abs x t) = join ["(λ", x, ".", show t, ")"]
show (App x y) = join ["(", show x, " ", show y, ")"]
show _ = error "Empty detected"
```

We can perform another step to improve the readability of the parser's specification: to each token (and token's data field) defined in Section 2.1, we can associate a short name to avoid unnecessary verbosity later on.

```
%token
    NL      { NewLine }
    VAR     { Term $$ }
    '.'     { Dot }
    'λ'     { Lambda }
    '('     { OpenPar }
    ')'     { ClosedPar }
```

We are finally ready to formally describe the language's grammar. Yet again, following Section's 1 set definition it is almost effortless to write a comprehensive description of λ-Calculus expressions: the only additional care we must take is to handle correctly multiple expressions in the same input file.

```
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 }
```

Before proceeding with the description of the evaluation strategies, it is worth to analyze two details about this definition:

1. Our specification requires the input stream to end with a NewLine token: we could solve this issue by introducing additional symbols and/or rewriting rules, but disambiguating the obtained grammar would require a lot of effort. Instead, we opted to "artificially" append a NewLine token at the end of the lexed stream before passing it to the parsing procedure, circumventing the issue efficiently and painlessly.

2. Contrary to the construction defined in Section 1, our specification allows the user to omit some brackets. This language expansion is a source of ambiguity, but we can tackle it by defining the abstraction and application as left-associative operations through the %left 'λ' '.' VAR directive. At the end we are still left with a few shift/reduce conflicts, but thankfully the defaulting rules of Happy produce a parser with the desired behaviour without additional work.

The generated parser will be the actual program compiled/interpreted, thus we need to define the main function of the software:

```
main :: IO ()
main = do
    s <- readFile "../test.txt"
    let tokens = alexScanTokens s ++ [ NewLine ]
    let parsedTerms = filter (/= Empty) (reverse (parse tokens))
    let evaluatedTerms = map eval parsedTerms
    mapM_ (print) evaluatedTerms
```

Let us continue by diving deeper into multiple possible implementations of the eval procedure.

2.3 Evaluating λ-terms: the algebraic approach

In this section we will introduce an algebraic approach to the problem of normalizing λ-terms: at its core, this strategy implements the β-reduction operation described in Section 1 to substitute terms within a suitable data structure. As previously explained, with this approach problems may arise due to involuntary name capture of free variables while normalizing, so we will elaborate on two different methods to overcome this issue.

2.3.1 Alpha conversion method

The first approach implements the α-conversion operation introduced in Section 1. For starters, let us illustrate the code of the eval function:

```
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
```

The process of β-reduction is actually applied in the first pattern of the definition: if a term consists in an application of an abstraction of variable x and body t to a term t_1, then substitute each (non-shadowed) occurrence of x in t with the normal form of t_1. The substitution routine sub will be introduced shortly. On the other hand, if the root of the input data structure is an application of a non-abstraction term to another term, normalize the first argument: if it reduces to an abstraction, apply recursively the first pattern, otherwise return an application of the two normalized arguments. Finally, the remaining patterns are required to evaluate recursively abstractions and to provide a base-case for the recursion.

Actually substituting a term within a term requires a slightly more complex algorithm:

```
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
sub x (App t1 t2) t = App (sub x t1 t) (sub x t2 t)
sub _ Empty _ = error "Empty detected"
```

The first pattern is the one which actually performs the substitution: if we have a variable denominated with the same symbol that must be substituted, then the variable is swapped-out in favour of the input term newVal, otherwise the expression is unchanged. This ensures that, for example, $x[x := y] \mapsto y$ and $x[z := y] \mapsto x$.

The second pattern actually introduces the α-reduction, as it deals with the substitution of a term within an abstraction. Let us explain the three cases separately:

1. If $x = y$, than it means that the symbol to substitute has been shadowed by a new declaration, thus the substitution must not been spread further. By doing that, we can guarantee that

$$(x(\lambda x.zx))[x := (\lambda y.y)] \mapsto ((\lambda y.y)(\lambda x.zx))$$
$$(x(\lambda x.zx))[x := (\lambda y.y)] \not\mapsto ((\lambda y.y)(\lambda x.z(\lambda y.y)))$$

2. The variable bound by the abstraction does not appear free within the input term, so we can apply the substitution without worrying about name clashes. Note that the function freeVars will be soon discussed.

3. In this case, $y \in FV(a)$, so we must apply the α-conversion procedure to obtain an abstraction that bounds another symbol (that does not belong in $FV(a)$), but with the same semantics. After renaming the variable, we can substitute recursively the applied parameter to the body of the updated function.

Finally, the third case only propagates the substitution through both children of an application. The last pattern only deals with potential misuse of the eval function (remember that we filtered out all the *Empty* objects right after parsing).

The specification of freeVars follows diligently the definition of FV presented in Section 1:

```
freeVars :: Term -> [String]
freeVars (Var x) = [x]
freeVars (Abs x t1) = removeDuplicates (freeVars t1) \\ [x] where
```

```
    removeDuplicates :: [String] -> [String]
    removeDuplicates [] = []
    removeDuplicates (x:xs)
        | elem x xs = removeDuplicates xs
        | otherwise = x : removeDuplicates xs
freeVars (App t1 t2) = freeVars t1 ++ freeVars t2
freeVars _ = error "Empty detected"
```

Where \\ is the list removal operator imported from Data.List.

Before concluding, we still need to explain how we engineered the α-conversion process. The renaming function (alphaConv) traverses the structure in a top-down fashion while substituting each occurrence of the problematic variable (clearly also the λ-bindings) with the new symbol.

```
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"
```

We still need to find a new, suitable, name. This task is executed by the function firstNameAvailable, called in a previous code snippet.

```
firstNameAvailable :: [String] -> String
firstNameAvailable = firstNameAvailableRec 0 where
    firstNameAvailableRec :: Int -> [String] -> String
    firstNameAvailableRec n usedVars
        | elem (variableSet !! n) usedVars = firstNameAvailableRec (n+1) usedVar
        | otherwise = variableSet !! n
```

This procedure returns the first occurrence of a variable that belongs to variableSet, but not to usedVars. From a Software Engineering perspective, one may be concerned by the fact that there is no error handling code: if all the variables within variableSet appeared in usedVars, we would end up trying to access an unexistent location of the former. Thankfully, Haskell features a lazy-evaluation mechanism that allows for the definition infinite structures. As a matter of fact, variableSet has been defined as a list, ordered by length (and, subsequently, alphabetically), of all the possible strings (thus we could visualize it as ["a", "b", "c", ..., "z", "aa", "ab" ..., "zz", "aaa", ...]): in practice, this guarantees that a new symbol will always be found.

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

Note that replicateM is the function imported from Data.List and [1..] is the infinite set of positive integers.

2.3.2 De Bruijn Indexes method

An alternative to α-conversion is implementing the De Bruijn Indexes. This consists in translating the λ-expressions into an alternative syntax where the variables bound by abstractions are not identified by their symbol, but instead by the level of nesting of their binding. There are multiple ways of representing free variables, but we chose to use directly the symbol with which they appeared in the original expression. Some example of this conversion are as follows:

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

By applying this conversion, we eliminate the need of binding names, so we can apply term substitutions within abstractions based solely on the order of application. To implement this strategy, we clearly need firstly a new data type that re-defines variables and abstractions:

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

instance Show DBTerm where
    show :: DBTerm -> String
    show (DBVar n) = Prelude.show n
    show (DBConst x) = x
    show (DBAbs t) = join ["(λ", show t, ")"]
    show (DBApp t1 t2) = join ["(", show t1, " ", show t2, ")"]
```

And we need a procedure capable of handling the translation:

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

This function traverses recursively the tree that defines the expression, gradually keeping track of the bound variables in the parameter vars. This ensures that, when we actually encounter a term in the form (Var x), we can establish whether x was a bound name (if there exists at least an index in corrispondence of which the symbol appears in vars), or if it is a constant, thanks to the

monadic characterization of elemIndex. Note that we managed vars as a stack instead of a list (we append items to the head, using the (:) operator) to allow name shadowing: by doing this, if elemIndex finds an integer, we are sure that it is the innermost binding of that variable.

After the translation, normalizing λ-terms in De Bruijn's notation becomes quite easier compared to the algebraic method with α-conversion. In fact, while the eval procedure remains practically the same, the sub routine is far less intricated, without the need of supporting functions:

```
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

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)
```

The sub function substitutes a variable with a given term only if the level of nesting of the variable is equal to the one of the searched binding (note that when we recursively iterate on an abstraction, we increase such number by 1, as we are "moving away" from the relevant binding). Clearly, constants never get substituted (notice how here we do not have the issue of name clashing with the free variables) and applications only need to spread the substitution through the tree's structure.

2.4 Evaluating λ-terms: the computational approach

While the α-conversion and the De Bruijn Indexes method are very similar at their core (firstly we guarantee to avoid name clashing through some form of transformation of the expression and then we apply a substitution), the computational method takes a completely different approach. It aims at exploiting Haskell's own anonymous functions to implement actual λ-abstractions capable of evaluating terms.

First of all, we need to define a new abstract data type capable of storing variables, applications and anonymous functions:

```
data CompTerm = CompVar String
```

```
  | CompAbs (CompTerm -> CompTerm)
  | CompApp CompTerm CompTerm
```

Since anonymous functions do not belong to the type classes Eq and Show, the whole data type will not be able to derive them either automatically. While being able to check terms for equality is not something we will need, printability is clearly crucial to ensure that we computed the right normal forms. We will explain our solution to this problem at the end of the section.

At first glance, this approach may seem very straightforward, as Haskell supports higher-order anonymous functions by default. The hidden issue in this technique lies in the closures: while we could manually write a function like $(\lambda x.(\lambda y.x))$ and exploit the language's λ-expression functionality to evaluate it, there is no straightforward way to nest recursively an unbounded number of expressions: in the previous example, we would not be able to build the innermost term as the constant x, as we would not know where said symbol was bound. Instead, we can implement our own elementary environments as lists of associations between names and parameters and use them to define recursively the λ-expressions:

```
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
        CompAbs f -> f t2'
        v@(CompVar _) -> CompApp v t2'
        a@(CompApp _ _) -> CompApp a t2'
    evalRec _ _ Empty = error "Empty detected"
```

Let us dissect the above code snippet to better understand how it works. The recursive procedure takes three parameters as input:

1. A list of actual parameters (initially empty)
2. A list of names for said parameters (initially empty)
3. An algebraic term to traverses

If the term is an abstraction, we create a new CompAbs instance that holds an actual anonymous function. Said function will be created recursively thanks to the environment augmented with the actual parameter (in this case, x) and the formal parameter (in this case, v). Note that, again, we build the closure as a stack to allow name shadowing later on.

If the term is a variable, then it must mean that we must distinguish whether it is bound or free. To do so, we must check if it appears within vars: if it does,

then we can return the corresponding term saved in the closure env; if not, that means that it is a constant and can be returned as is.

If the term is an application, then we have to check the type of its first argument: if it is an abstraction, then we can straightforwardly apply it to the normalized parameter t_2'. On the other hand, if it is an application or a variable, we can always reduce the second argument and then save the normalized application.

By substituting the algebraic terms with actual functions we avoid the issue of name clashing, since when an abstraction will be ready for application, it will evaluate its argument according to the environment indexing, in a similar fashion to De Bruijn's Indexes.

As introduced before, printing an expression will not be as simple as in the previous approaches: it is not possible to inspect an anonymous function's body after creation, which essentially becomes an inspetionable *black-box*. Our best bet is to deduce the structure of a λ-expression by passing selected parameters to it and looking how it computes the result: we can not scrutinize the computation steps by themselves, but the instances of CompVar can. For example, if we had a function f and we knew that $fa = a$, then we could infere that either $f = \lambda x.x$ or $f = \lambda x.a$. We must find a way to discriminate between such cases: by coming back to the definition of our lexer in Section 2.1, we can see that we did not allow numbers as identifiers, only alphabetic characters. If we fed numerical arguments (as strings, of course), to the functions, we could easily distinguish between actual bound variables (that would end up being numberical) and constants (which will belong to the set of recognised tokens, i.e. letters). This procedure can be applied recursively to deduce the structure of the abstractions at each nesting level.

In practice, we coded a function that transforms a CompTerm into a Term by feeding strings encoding numbers incrementally larger into the abstractions and observing the results:

```
toSyntax :: CompTerm -> (Term, [String])
toSyntax = toSyntaxRec 0 where
    toSyntaxRec :: Int -> CompTerm -> (Term, [String])
    toSyntaxRec _ (CompVar x) = case readMaybeInt x of
        Just n -> (Var (show n), [])
        Nothing -> (Var x, [x])
    toSyntaxRec n (CompAbs f) = let (term, freeVars) = toSyntaxRec (n+1) (f (Com
    toSyntaxRec n (CompApp t1 t2) = let
                (t1', freeVars1) = toSyntaxRec n t1
                (t2', freeVars2) = toSyntaxRec n t2
            in (App t1' t2', removeDuplicates (freeVars1 ++ freeVars2)) where
                removeDuplicates :: [String] -> [String]
                removeDuplicates [] = []
                removeDuplicates (x:xs)
                    | elem x xs = removeDuplicates xs
                    | otherwise = x : removeDuplicates xs
```

```
readMaybeInt :: String -> Maybe Int
readMaybeInt = readMaybe
```

In the base case, toSyntaxRec discriminates between bound (values that can be read as an integer) and free (values that encode strings) variables. Starting from this, it constructs the set of free variables with a top-down approach: applications only need to spread the conversion further, functions must feed a number and increment it gradually as explained before. In the end we end up with a pair that contains the algebraic expression as first coordinate and the set of free variables as the second.

Now we only need to substitute the numeric symbols with alphabetical names while being careful of not causing accidental name captures. The following function does exactly that:

```
substInts :: (Term, [String]) -> Term
substInts (term, freeVars) = substIntsRec [] term freeVars where
    substIntsRec :: [String] -> Term -> [String] -> Term
    substIntsRec varNames (Var x) _ = case readMaybeInt x of
        Just n -> Var (varNames !! n)
        Nothing -> Var x
    substIntsRec varNames (App t1 t2) usedVars = App (substIntsRec varNames t1 u
    substIntsRec varNames (Abs _ t) usedVars = let x = firstNameAvailable usedVa
    substIntsRec _ Empty _ = error "Empty detected"
```

We substitute the bound variables and the bindings while being careful to avoid to choose the free variables (note the use of the previously defined function firstNameAvailable).

3 Conclusions

In this report, we implemented different techniques to evaluate λ-expressions. The algebraic approach is generally simpler to realize, especially in Haskell where efficient recursion and pattern matching make the implementation of the substitution procedure almost trivial. This approach may suffer of the problem of name clashing: to avoid it, we analyzed both the α-conversion and the De Bruijn Indexes techniques. The former requires some care to handle the names, but yet again Haskell's capability of handling infinite structures through lazy evaluation proves to be useful as we have access to an unlimited set of symbols. The latter, on the other hand, requires a conversion at first, but the substitution becomes effortless after. We decided to keep the expressions in De Bruijn's notation to provide some examples of this alternative, but it would be easy to write a function that translates the terms back into the original notation if we wanted to. Finally, we showed how it is possible to normalize expressions exploiting Haskell's own anonymous functions: this approach, although seems easy at first, quickly becomes slightly cumbersome as it requires to create en-

vironments and to develop a quite complicated print mechanism with multiple conversions.

As future work, it would be interesting to see if there are simpler, more elegant alternatives to the double-translation printing procedure for the computational approach and, more generally, if it is possible to tackle the problem of normalizing λ expressions with alternative methods.