Simulation of Lambda Terms in Lambda Calculus

Helmut Brandl

(firstname dot lastname at gmx dot net)

Version 1.0

Abstract

In this short paper we develop a function in lambda calculus which normalizes another lambda term given as an argument. I.e. we simulate computation of lambda calculus (i.e. beta reduction) within lambda calculus.

The normalizing function needs only one partial function to do the job. All other used functions are total. The same key idea is used when proving that all computable functions have a *Kleene Normal Form*.

In this paper a programming notation for lambda calculus is used in order to make the subject accessible for programmers and not only for mathematicians.

For comments, questions, error reporting feel free to open an issue at `https://github.com/hbr/Lambda-Calculus`

Contents

1 Introduction

In this paper we develop a lambda term which can evaluate an arbitrary other lambda term. This is similar to the *universal turing machine* which can simulate an arbitrary turing machine.

Why is this interesting?

Firstly it an interesting programming exercise. It might be surprising that it is not very difficult to write a simulator of lambda terms in lambda calculus.

Secondly it is a demonstration of the computational power of lambda calculus. Lambda calculus can evaluate itself.

Thirdly it shows that any computation can be transformed into something similar to *Kleene's normal form* of computation which basically says that any computable function needs only one partial function.

Let's look into the last point a little bit.

In the beginning of the 20th century 3 definitions of computatable functions had been given. Kurt Gödel's recursive functions, Alan Turing's automatic machines (today called Turing machines) and Alonzo Church's lambda calculus. It has been proved that these definitions are equivalent i.e. for each definition of a computable function in one formalism and equivalent definition in the other formalisms can be found.

In all three formalisms you can define functions which are total (i.e. terminate for all input values) and which are partial (i.e. are not guaranteed to terminate for all input values).

For recursive functions Stephen Kleene has shown that any computable function can be transformed into another computable function which has only one partial subfunction. For recursive functions the total functions are the primitive recursive functions. Partial functions include μ-recursive functions which search the minimal natural number satisfying a certain predicate. Since such a minimal number might not exist, μ-recursive functions are not guaranteed to terminate. μ-recursive functions are the equivalent of *while* loops in imperative programming.

For Turing machines Alan Turing invented the universal (Turing) machine which contains only one potentially not halting submachine all other submachines being guaranteed to halt on any valid input.

Since all models of computation are equivalent it must be possible to construct a *universal* lambda term which is able to do any computation in lambda calculus using only one partial function.

In order to define such a universal lambda term we go step by step.

In section 2 we give a compact introduction to lambda calculus. This section introduces the basic definitions of lambda calculus, the way computation is done via beta reduction, it defines normal forms and shows that normal forms are unique via the Church Rosser theorem and finally it introduces leftmost reduction as a canonical way to reduce lambda terms. Those of you familiar with lambda calculus can skip this chapter.

The next section 3 introduces a programming notation for lambda calculus. Using the mathematical notation for lambda calculus can be very tedious. Programming notation in ascii is much more readable and gives the connection to functional programming.

In the programming notation section we introduce the encoding of boolean values, pairs, optional values and natural numbers and some basic functions on these values. All lambda terms introduced in this

section are total in the sense that beta reduction finally converts the terms into their corresponding normal forms which are guaranteed to exist.

In the section 4 we introduce a lambda term we represents a partial function. This is the only partial function we need to construct the simulator of a lambda term in lambda calculus.

Within section 5 a nameless representation of lambda terms is introduced which is equivalent to the standard definition of lambda terms given in the preliminaries 2. The nameless representation has the advantage that it avoids any need to rename variables because of clashes between local variable names and variables defined in an outer context.

The nameless representation makes the construction of the lambda simulator simpler because it avoids the nasty topic of renaming variables.

The nameless representation just needs a shift operator and a definition of beta-reductions based on the shift operator. All other definitions are same as in the standard representation.

The core of the paper is section 6 which constructs a lambda term which can evaluate other lambda terms.

In order to evaluate lambda terms we have to *encode* the lambda terms. This is necessary, because lambda calculus cannot inspect lambda terms given in its raw form. An *encoding* for lambda terms is shown in a manner such that any the encoding of any lambda term is straightforward.

Note that Kleene's normal form and Turing's universal machine use encodings as well. Both have chosen to transform the recursive function or the Turing machine respectively into a corresponding Gödel number. We could have used Gödel numbers as well. But the encoding given in this paper seems to be more natural and uses the basic idea of algebraic data types of functions languages.

To demonstrate the connection of the encoding of lambda terms in lambda calculus a similar encoding of lambda terms in the programming language Ocaml is given. This is just for illustrative purposes.

In the next steps we construct recursive functions on encoded lambda terms, redex reduction and leftmost reduction on encoded lambda terms.

Finally we define a function which reduces any encoded lambda term into its normal form of loops forever in case the normal form does not exist. The normalization function is the only function which uses

the partial function defined in section 4. The normalization function is just an iteration of a leftmost reduction step which terminates when there is no more leftmost reduction step possible i.e. when the term is in normal form.

2 Preliminaries

This section contains a short introduction into lambda calculus and the used notation. The details can be studied in the papers [1], [2] or [3].

The notion of inductively defined sets and relations and the corresponding proof methods are described in detail in [1] and in [4].

For those familiar with lambda calculus and its notations the section can be skipped.

2.1 Basic Definitions in Lambda Calculus

Definition 2.1. A *lambda term* is either a variable name (from a countably infinite set of variable names), an application of a term to another or a lambda abstraction. It is defined by the grammar

$$\begin{array}{rcll} t & := & x & \text{variable} \\ & | & tt & \text{application} \\ & | & \lambda x.t & \text{lambda abstraction} \end{array}$$

Application is left-associative i.e. abc is parsed as $(ab)c$ and $\lambda xy.e$ is used as an abbreviation for $\lambda x.\lambda y.e$.

Definition 2.2. A *redex* (*red*ucible *ex*pression) is a lambda term of the form

$$(\lambda x.e)a$$

$\lambda x.e$ is called the *function term* of the redex and a is called the *argument term* of the redex.

The expression is called *reducible* because it can be reduced in one step to

$$e[a/x]$$

which is the body e where all free occurrences of the variable x are replaced by the argument a. Note that it might be necessary to rename some of the bound variables in the body e such that there is no collision with the free variables in the argument term a (i.e. avoid variable capture).

Reducing a redex is a computation step in lambda calculus. Since more than one redex can be contained within a lambda term, the next computation step is not unique. Usually different steps can be made. Therefore the basic reduction step, a *beta reduction* is not a function, it is a relation relating the term before and after the reduction.

Definition 2.3. *Beta Reduction*: $t \xrightarrow{\beta} u$ says that the term t reduces to the term u by reducing one of the redexes in t. The relation is defined inductively by the rules

1. $(\lambda x.e)a \xrightarrow{\beta} e[a/x]$
2. $$\frac{a \xrightarrow{\beta} t}{ab \xrightarrow{\beta} tb}$$
3. $$\frac{b \xrightarrow{\beta} v}{ab \xrightarrow{\beta} av}$$
4. $$\frac{t \xrightarrow{\beta} u}{\lambda x.t \xrightarrow{\beta} \lambda x.u}$$

We write $a \xrightarrow{\beta^*} b$ if a reduces to b in zero or more steps.

The reduction of a lambda abstraction results in a lambda abstraction because of the fourth rule. However due to the first rule the reduction of an application can be anything depending on the body of function term and the argument. Therefore we define a *base term* which is an application maintaining its structure during reduction.

Definition 2.4. A *base term* is a variable applied to zero or more terms. The set of all base terms BT is defined inductively by the rules:

1. $x \in \mathrm{BT}$
2. $$\frac{t \in \mathrm{BT}}{tu \in \mathrm{BT}}$$

A computation in lambda calculus is finished if there is no more redex in a term.

Definition 2.5. A term t is in *normal form* if it does not contain a redex. We call NF the set of terms in normal form. It is defined inductively by the rules

1. $x \in \mathrm{NF}$
2. $$\frac{t \in \mathrm{NF} \quad t \in \mathrm{BT} \quad u \in \mathrm{NF}}{tu \in \mathrm{NF}}$$
3. $$\frac{e \in \mathrm{NF}}{\lambda x.e \in \mathrm{NF}}$$

Note that in the second rule the condition $t \in \mathrm{BT}$ is essential. Otherwise t could be a lambda abstraction in normal form and tu would be a redex which is not in normal form by definition.

It is evident from the rules that a term in normal form is either a base term or a lambda abstraction. Furthermore all subterms of a term in normal form are in normal form as well.

Note that it is perfectly possible that a lambda term cannot be reduced to normal form. Consider the term $\omega := \lambda x.xx$. The term ωa is a redex which reduces to aa. If we apply ω to itself we get an infinite reduction sequence.

$$\begin{aligned} \omega\omega &= (\lambda x.xx)\omega \\ &\xrightarrow{\beta} \omega\omega \\ &\xrightarrow{\beta} \omega\omega \\ &\xrightarrow{\beta} \ldots \end{aligned}$$

A similar divergence is possible with the term $U := \lambda xf.f(xxf)$.

$$\begin{aligned} UUf\ldots &= (\lambda xf.f(xxf))Uf\ldots \\ &\xrightarrow{\beta} f(UUf)\ldots \\ &\xrightarrow{\beta} f(f(UUf))\ldots \\ &\xrightarrow{\beta} \ldots \end{aligned}$$

As opposed to the term $\omega\omega$ the term $UUf\ \ldots$ might have a normal form if the function f depending on the arguments $\ldots$ throws away its first argument UUf which is responsible for the possible divergence.

Beside the notion of *beta reduction* we need the notion of *beta equivalence*.

Definition 2.6. Two terms t and u are *beta equivalent* expressed as $t \overset{\beta}{\sim} u$ if t can be transformed into u using zero or more forward or backward beta reduction steps. The relation $\overset{\beta}{\sim}$ is defined inductively by the rules

1. Reflexivity: $t \overset{\beta}{\sim} t$
2. Forward: $\dfrac{t \overset{\beta}{\sim} u \quad u \overset{\beta}{\to} v}{t \overset{\beta}{\sim} v}$
3. Backward: $\dfrac{t \overset{\beta}{\sim} u \quad v \overset{\beta}{\to} u}{t \overset{\beta}{\sim} v}$

2.2 Confluence - Church/Rosser

As seen in the definition of beta reduction 2.3 a beta reduction step is not unique, because a term a might contain more than one redex. I.e. $a \overset{\beta}{\to} b$ and $a \overset{\beta}{\to} c$ is possible for different terms b and c.

If we want to use lambda calculus as a model of computation, then at least the final result (the normal form, if it exists) shall be unique. In order to prove that, the Church Rosser theorem is necessary.

Theorem 2.7. *Church Rosser theorem*: If a reduces to b in zero or more steps ($a \overset{\beta^*}{\to} b$) and a reduces to c in zero or more steps ($a \overset{\beta^*}{\to} c$), then there exist a term d which is a common reduct of b and c.

$$\begin{array}{ccc} a & \overset{\beta^*}{\to} & b \\ \downarrow_{\beta^*} & & \downarrow_{\beta^*} \\ c & \overset{\beta^*}{\to} & \exists d \end{array}$$

I.e. even if a reduction sequence goes via different paths, there is always a common reduct where the paths meet. Because of this theorem the lambda calculus is characterized as *confluent*.

Proof. See e.g. [1] or [3]. □

The *Church Rosser theorem* has the following important corollaries that normal forms are unique and beta equivalent terms have common reducts.

Corollary 2.8. *Uniqueness of Normal Forms*: If $t \overset{\beta^*}{\to} u$ and u is in normal form, then this normal form is unique.

Proof. Assume that u and v are different normal forms of t. Because of $t \overset{\beta^*}{\to} u$ and $t \overset{\beta^*}{\to} v$ and the Church Rosser theorem there exist a common reduct of the terms u and v. Since u and v are in normal form they only reduce to themselves in zero steps i.e. they have to be identical. □

Corollary 2.9. *For two beta equivalent terms* $t \overset{\beta}{\sim} u$ *there exists a common reduct* v *such that* $t \overset{\beta^*}{\to} v$ *and* $u \overset{\beta^*}{\to} v$.

Proof. If t and u are the same terms, then the common reduct is the term itself.

If t and u are not the same terms then by the definition of beta equivalence 2.6 there is always an equivalence path of one of the forms

$$t \overset{\beta}{\sim} a \overset{\beta}{\to} b \overset{\beta}{\sim} u$$
$$t \overset{\beta}{\sim} b \overset{\beta}{\leftarrow} a \overset{\beta}{\sim} u$$

For the forward step assume as an induction hypothesis that t and a have the common reduct c. Then the Church Rosser theorem guarantees the existence of a common reduct d of b and c which is also a common reduct of t and b.

For the backward step assume as an induction hypothesis that t and b have the common reduct d. By definition of beta reduction 2.3 d is also a reduct of a. Therefore d is a common reduct of t and a. □

2.3 Leftmost Reduction

Beta reduction 2.3 is not unique because there might be more than one redex which can be contracted. In order to have a unique reduction we define *leftmost reduction* which contracts the leftmost redex in a lambda term.

Definition 2.10. *Leftmost reduction*: We write $a \overset{\beta_{\text{lm}}}{\to} b$ if a reduces to b by reducing the leftmost redex in a. The relation $\overset{\beta_{\text{lm}}}{\to}$ is defined inductively by the rules

1. $(\lambda x.e)a \overset{\beta_{\text{lm}}}{\to} e[a/x]$
2. $$\frac{t \overset{\beta_{\text{lm}}}{\to} u \quad t \text{ is an application}}{tv \overset{\beta_{\text{lm}}}{\to} uv}$$
3. $$\frac{t \in \text{NF} \quad t \in \text{BT} \quad u \overset{\beta_{\text{lm}}}{\to} v}{tu \overset{\beta_{\text{lm}}}{\to} tv}$$
4. $$\frac{t \overset{\beta_{\text{lm}}}{\to} u}{\lambda x.t \overset{\beta_{\text{lm}}}{\to} \lambda x.u}$$

The second rule guarantees that a redex in the head position is reduced. The third rule guarantees that only a leftmost redex can be reduced (all terms to the left of u are base terms in normal form.

Leftmost reduction is unique because all 4 rules are mutually exclusive.

Theorem 2.11. *Leftmost reduction is normalizing* If there is a reduction $t \xrightarrow{\beta^*} u$ and u is in normal form, then the normal form u can be found by zero or more leftmost reduction steps $t \xrightarrow{\beta^*_{\text{lm}}} u$.

Proof. See chapter 11.4 in [3] □

3 Programming in Lambda Calculus

3.1 Programming Notation

The mathematical notation described in section 2.1 is not very convenient to express complicated lambda terms. In order to handle complex lambda terms more conveniently we use a programming language notation as defined in [2].

The correspondence between lambda terms in mathematical notation and lambda terms in programming notation is given by the following table

x	`x`
ab	`a b`
$\lambda x.e$	`\ x := e`

The symbol λ is replaced by the ascii backslash \ like in the programming language haskell and the dot is replaced by the *"is defined as"* symbol `:=` .

Some examples:

$\lambda xy.x$	`\ x y := x`
$\lambda xyf.fxy$	`\ x y f := f x y`

Furthermore we allow definitions like

```
true := \ x y := x
```

where the name `true` is defined as the lambda term `\ x y := x`. Note that `:=` is right associative.

Definitions have to be acyclic i.e. recursion is not allowed. The lambda term is always considered as the lambda term where all definitions are expanded.

We define functions more handy as

```
true x y := x
```

which we consider equivalent to the above definition.

Inside an abstraction local definitions are allowed

```
fst p :=
    p f where
        f x y := x

-- which is equivalent with
fst := \ p := p (\ x y := x)
```

Local definitions like definitions have to be acyclic i.e. the lambda term is considered as the lambda term where all local definitions are expanded.

In the following sections we give definitions of commonly used functions in programming notation. The sections are kept short and concise. For more details look into [2].

3.2 Booleans

Boolean values are represented in lambda calculus as functions taking two arguments. The boolean value `true` returns the first argument and the boolean value `false` returns the second argument. Conjunction, disjunction and negation can be defined easily.

```
true  x y := x
false x y := y

and a b := a b false
or  a b := a true b
not a   := a false true
```

As described above the symbol `true` is just an abbreviation for the lambda term $\lambda xy.x$ and the symbol `false` is just an abbreviation for the lambda term $\lambda xy.y$. Both terms are in normal form.

If the terms `true` and `false` are applied to two arguments `a` and `b` then they return the first or the second argument. As long as the arguments are normalizing (i.e. reduce in zero or more steps to normal

form) then the terms `true a b` and `false a b` are normalizing as well.

The same applies to any term `f a b` as long as `f` reduces in zero or more steps to `true` or `false`.

3.3 Pairs

The function `pair` takes three arguments. A pair of the terms `a` and `b` is obtained by applying the function only to these arguments `pair a b`. The first and second element of a pair can be obtained by providing the third argument.

```
pair a b f := f a b
fst  p     := p (\ a b := a)
snd  p     := p (\ a b := b)
```

3.4 Natural Numbers

Natural numbers are usually represented in lambda calculus by their Church encoding as Church numerals.

A Church numeral is a function taking two arguments `f` and `s`. It applies the term `f` n times on the argument `s` in order to represent the natural number n.

```
zero    f s  := s
succ n  f s  := f (n f s) -- or n f (f s)

one := succ zero
two := succ one
...
```

The function `is-zero` applies the function `\ x := false` n times upon the start term `true`. For the Church numeral `zero` the value `true` is returned and for any other numeral the value `false` is returned.

Addition of the numerals `a` and `b` just applies `a` times the function `succ` on the start value `b`.

```
is-zero n :=
    n (\ x := false) true

(+) a b := a succ b
```

In order to do recursion with Church numerals we need a trick. We compute pairs where the first component is the argument and the second component is the value of the function. Finally the argument is thrown a way.

The iteration starts with `pair zero s` where `s` is the result of the recursive function for the number zero. The iteration function has to decompose the pair consisting of the predecessor number and the predecessor result and compute the pair consisting of the actual number and its result.

```
nat-rec n f s
    -- Do recursion with the function 'f' where 'f' takes
    -- as a first argument the predecessor number and as
    -- a second argument the result of the predessor and
    -- computes the result of the actual number.
    -- Recursion starts with 's' for the number zero.
:=
    snd (n step (pair zero s)) where
      step p :=
        p (\ n0 r0 := pair (succ n0) (f n0 r0))
```

Having the recursor `nat-rec` many other functions can be defined.

```
pred n :=
    nat-rec (\ n0 r0 := n0) zero

(-) a b := b pred a

(<=) a b := is-zero (a - b)

(=)  a b := (a <= b) and (b <= a)

(<)  a b := (succ a <= b)

lt-eq-gt a b x y z
    -- if 'a < b' then return 'x',
    -- if 'a = b' then return 'y',
    -- if 'a > b' then return 'z'
:=
        -- Note 'not (a <= b)' implies 'a > b'
        --      'not (b <= a)' implies 'a < b'
        --      '(a <= b)' and '(b <= a)' implies 'a = b'
```

```
(a <= b) ((b <= a) y x) z
```

3.5 Optional Values

A lambda term representing an optional value takes two arguments. The first argument is a function which is applied to the optional value if present. The second argument is the return value in case no optional value is present.

```
some x   f n :=  f x
none     f n :=  n
```

4 Partial Functions

All functions introduced in the previous chapter 3 are total functions in the following sense: As long as appropriate arguments are given (all values strongly normalizing and all functions applied to appropriate arguments are strongly normalizing) the resulting values are strongly normalizing as well.

However sometimes functions are needed which compute a certain value but are not guaranteed to terminate. I.e. even when applied to appropriate arguments they might not reach a normal form.

In imperative programming there are while loops which repeat a loop as long as a certain condition is valid and it is not guaranteed that a state is reached where the condition of the while loop is no longer valid i.e. the while loop terminates.

In this chapter we develop a function in lambda calculus which corresponds to a while loop in imperative programming. We look for a function `fixpoint f s` with the following specification:

- Start with the value s.
- The function f takes a value and returns an optional value (see 3.5).
- Iterate the function f on the start value as long as it returns `some v`.
- As soon as fv returns `none` the value v is the result of `fixpoint f s`.

The function is partial in the sense that it might be possible that fv never returns `none` in the course of the iteration.

In order to do a possibly unlimited iteration we use the function

```
U g x := g (x x g)
```

which has the following reduction behaviour

```
U U g s    =    (\ g x := g( x x g)) U g s
           ~>   g (U U g) s
```

The function g when applied to the arguments UUg and s has the possibility to return one of

1. `U U g s1`
2. `r`

In the first case the iteration continues with the next iteration value `s1`. In the second case the result `r` is returned.

```
fixpoint f s :=
    U U g s where
        g z v :=
            f v z v

        U g x :=
            g (x x g)
```

To verify the correctness let's see how the iteration works

```
U U g v

~> g (U U g) v

~> f v (U U g) v
```

where v is initially the start value s.

Now there are two possibilities:

1. `f v` returns `some v1` where v_1 is the next iteration value. In this case `f v (U U g) v` returns `U U g v1` and the iteration continues.
2. `f v` returns `none`. In this case `f v (U U g) v` returns `v`

5 De Bruijn Indices

5.1 Nameless Representation

The names of bound variables are not important. Terms which differ only in the names of the bound variables are equivalent. E.g. the terms $\lambda x.x$ and $\lambda y.y$ are equivalent and the terms $\lambda xy.x$ and $\lambda ab.a$ are equivalent. The terms $\lambda xy.x$ and $\lambda ab.b$ are not equivalent.

Furthermore it is important to avoid interference of free and bound variables i.e. to avoid variable capture. This requires sometimes to rename bound variables such that their names are different from free variables or from variables bound by outer binders.

These inconveniences can be avoided by using De Bruijn indices as variable names. The lambda terms with De Bruijn indices are formed according to the grammar

$$\begin{array}{rcll} t & := & j & \text{De Bruijn index} \\ & | & tt & \text{application} \\ & | & \lambda t & \text{lambda abstraction} \end{array}$$

This is a nameless representation. The nameless representation of a lambda term can be obtained from the named representation by using De Bruijn indices instead of variable names. The De Bruijn index of a variable is obtainded by the following:

- Bound variable: The De Bruijn index j of the variable x is the number of binders between the occurrence of the variable x and its closest binder λx looking inside out.
- The De Bruijn index of the free variable x is $j + n_B$ where j is the position of its first occurrence in a left to right scan and n_B is the number of binders at the occurrence of the free variable looking inside out.

Usually nameless representations are only needed for combinators (combinators are lambda terms with no free variables).

Some examples:

$\lambda x.x$	$\lambda 0$	No binder between x and λx
$\lambda xy.y$	$\lambda\lambda 0$	No binder between y and λy
$\lambda xy.x$	$\lambda\lambda 1$	One binder between x and λx
$x(\lambda y.xyz)$	$0(\lambda 102)$	x and z are free variables with the left to right positions 0 and 1. The first occurrence of x is not within a binder. The second occurrence of x and the first occurrence of z are within one binder.

I.e. there is no one to one mapping between a variable name and its De Bruijn index. The De Bruijn index of the same variable is different if there are a different number of binders between the occurrence of a variable and its closest binder.

5.2 Shift Operator

In order to put a term t outside a binder into the binder we have to adapt the De Bruijn indeces used in the term. The index i outside the binder becomes $i+1$ inside the binder. However if i is already bound within t it must not be changed.

More generally we have to able to put a term t inside n more binders. In order to do this we define a shift operator.

The shift operator $\uparrow_b^n$ shifts all De Bruijn indices in a term t up by n starting from the cutoff index b and leaves all indices strictly below the cutoff index unchanged.

$$\uparrow_b^n t := \begin{cases} \uparrow_b^n i & := \begin{cases} i & \text{if } i < b \\ i+b & \text{otherwise} \end{cases} \\ \uparrow_b^n (tu) & := (\uparrow_b^n t)(\uparrow_b^n u) \\ \uparrow_b^n (\lambda t) & := \lambda(\uparrow_{b+1}^n) \end{cases}$$

In order to put a term t in an environment with n more binders we write $\uparrow_0^n t$.

5.3 Beta Reduction

In the representation with names the redex $(\lambda x.e)a$ is reduced to $e[a/x]$ where all free occurrences of the variable x in e are replaced by the argument term a.

In the nameless representation with De Bruijn indices a redex has the form $(\lambda e)a$. The De Bruijn index of the variable of the reduced

binder is 0 (or $0 + b$ if the variable x occurs within more binders than the toplevel binder). I.e. depending on how deeply nested we have to replace the De Bruijn index $0 + b$ by the argument a.

Let's assume we replace the variable with the index i by the argument term. I.e. between the reduced binder λe and the actual occurrence of the variable there are i binders.

All indices strictly below i are bound by binders inside the reduced binder. These indices are left unchanged.

All indices strictly above i are either free variables or bound by binders outside the reduced binder. They go down by one, because one binder is removed.

The index i represents the variable of the reduced binder. It has to be replaced by the argument a. But the argument a is valid at the same binding level as the reduced lambda abstraction λe (the redex is $(\lambda e)a$. By replacing the index i it *enters* i more binders. Therefore we have to replace the De Bruijn index i by $\uparrow_0^i a$.

The beta reduction in nameless form reads

$$(\lambda e)a \xrightarrow{\beta} e[a/0]$$

using the following recursive definition of $e[a/i]$

$$e[a/i] := \begin{cases} j[a/i] & := \begin{cases} j & \text{if } j < i \\ \uparrow_0^i a & \text{if } j = i \\ j - 1 & \text{if } j > i \end{cases} \\ (tu)[a/i] & := t[a/i]\ u[a/i] \\ (\lambda t)[a/i] & := t[a/(i+1)] \end{cases}$$

6 Simulator

In this chapter we develop a lambda term `normalize` which computes the normal form of a term, if it exists, and loops forever, if no normal form exists.

It is not possible to feed a lambda term directly into the function `normalize`. We have to find an encoding of a lambda term which allows the function to inspect the term.

Furthermore we do not want to struggle with variable renamings, therefore we use the canonical form of lambda terms with De Bruijn indices.

6.1 Encoding of Lambda Terms in Lambda Calculus

We use the canonical form of lambda terms with De Bruijn indices which are formed according to the grammar as described in the chapter 5.1

$$\begin{array}{rcll} t & ::= & i & \text{De Bruijn index of the variable} \\ & | & tt & \text{application} \\ & | & \lambda t & \text{lambda abstraction} \end{array}$$

We need 3 constructors to encode an arbitrary canonical lambda term as a lambda term. One constructor for variables (i.e. De Bruijn indices), one for applications and one for lambda abstractions.

An encoded lambda term is a function with 4 arguments:

1. b: Church numeral representing the number of bound variables.
2. v: Function transforming the Church numeral of the De Bruijn index of the variable and the number of bound variables into the result term of the variable.
3. a: Funktion transforming the result terms of the function term and the argument term of an application into the result term of the application.
4. l: Function transforming the result term of the body of the abstraction into the result term of the lambda abstraction.

We assign types to the arguments of the encoded lambda term (note that the types are just for our understanding of the arguments; the lambda calculus is untyped)

```
b: Nat                  -- 'Nat' is a Church numeral

v: Nat -> Nat -> R      -- 'R' is the type of the result

a: R -> R -> R

l: R -> R
```

An encoded lambda term has therefore the type

```
Lam = ( Nat ->
        (Nat -> Nat -> R) ->
        (R -> R -> R) ->
        (R -> R) ->
```

```
      R
    )
```

Now it is easy to design the 3 necessary constructors.

```
var k      b v a l  :=  v k b

app x y    b v a l  :=  a (x b v a l) (y b v a l)

lam x      b v a l  :=  l (x (succ b) v a l)
```

1. The constructor for a variable with De Bruijn index k is a function mapping the 4 arguments into an application of the argument v to the index k and the number of the bound variables b (note that numbers are encoded as Church numerals).
2. The constructor for an application of the function term x to the argument y is a function mapping the 4 arguments into an application of the argument a onto the results obtained from the function term $xbval$ and the argument term $ybval$.
3. The constructor for a lambda abstraction with body x is a function mapping the 4 arguments into an application of the argument l onto the result obtained from the body applied to the arguments x (succ b) v a l. Note that the number of bound variables has to be increased by one because the lambda abstraction binds one more variable.

We show for the lambda terms $\lambda x.x$ and $\lambda xy.x$ the mathematical notation, the notation in program form, the mathematical notation of the canonical form with De Bruijn indices and the encoded term in program notation.

Math	Progr Notation	De Bruijn	Encoded
$\lambda x.x$	`\ x := x`	$\lambda 0$	`lam (var zero)`
$\lambda xy.x$	`\ x y := x`	$\lambda\lambda 1$	`lam (lam (var one))`

Note that the term in program notation and the encoded term in program notation are different (to get the complete picture you have to expand the abbreviations of `lam`, `var`, `one` and `zero`). I.e. the encoded term is much more complex than the original term. The added complexity makes it possible to inspect the structure of a lambda term within lambda calculus.

In order to demonstrate the handling of encoded lambda terms we show a function which computes the number of subterms (including

the term) of a lambda term. The number of subterms is defined recursively. A variable has one subterm (the variable itself), an application has one more subterm than the sum of the subterms of the function and the argument term and an abstraction has one more subterm than the number of subterms of the body.

```
size t :=
    t
        zero
        (\ k b := one)
        (\ x y := succ (x + y))
        (\ x := succ x)
```

6.2 Encoding in e.g. Ocaml

An encoding of a lambda term in lambda calculus is just an iteration over the lambda term. In order to get more familiar with this type of encoding we compare it with an encoding in the language Ocaml and demonstrate that the above shown encoding is just an implementation of a fold function in Ocaml.

A lambda term in nameless form could be encoded in Ocaml as the algebraic type

```
type lam =
    | Var of int
    | App of lam * lam
    | Lam of lam
```

Having this we can define a recursive function `fold` which does the same thing as the above encoding in ocaml.

```
let rec fold (t: lam) (b: int)
     (v: int -> int -> 'r)
     (a: 'r -> 'r -> 'r)
     (l: 'r -> 'r) : 'r =
    match t with
    | Var k ->
        v k b
    | App (t, u) ->
        a (fold t b v a l) (fold u b v a l)
    | Lam t ->
        l (fold t (b + 1) v a l)
```

Note that the partial call `fold t` has the same type as a lambda term encoded in lambda calculus.

6.3 Recursion

The encoding of a lambda term iterates over the structure of the encoded lambda term in a bottoms up manner. The bottom terms are the variables where the argument v computes the result term for the variable. The intermediate terms representing an application or a lambda abstraction can use the computed results of the subterm(s) and compute the result for the corresponding term.

In a real recursion the functions doing the work of the intermediate terms (application and lambda abstraction) need not only access to the result of the subterm(s) but also to the encodings of the subterm(s). In order to achieve this type of recursion in lambda calculus we use the trick that the recursion not only computes the result of the corresponding term. It computes a pair consisting of the term and the corresponding result. At the end we throw away the encoded term (similar to the recursor `nat-rec` in section 3.4).

The 4 arguments b, v, a and l in a real recursion have the types

```
b: Nat                  -- Bound vars
v: Nat -> Nat -> R      -- De Bruijn -> Bound vars -> R
a: Lam -> R -> Lam -> R -> R
l: Lam -> R -> R
```

where `Lam` is the type of the encoded lambda terms. The function doing recursion has the structure

```
lam-rec t b v a l :=
    snd (t b v0 a0 l0) where
        v0 k b    :=  ...
        a0 p1 p2  :=  ...
        l0 p1     :=  ...
```

The function v_0 takes the De Bruijn index k and the number of bound variables b (both encoded as Church numerals) and returns the pair consisting of the encoding of the variable with De Bruijn index k and the result.

```
v0 k b :=
    pair (var k b) (v k b)
```

The function a_0 takes the two pairs p_1 and p_2 and returns a pair consisting of the encoding of the application and the result of the application

```
a0 p1 p2 :=
    p1 (\ t1 r1 :=
            p2 (\ t2 r2 :=
                    pair (app t1 t2) (a t1 r1 t2 r2)
               )
       )
```

The function l_0 takes the pair p and returns the pair consisting of the lambda abstraction and the result of the lambda abstraction

```
l0 p1 :=
    p1 (\ t r := pair (lam t) (l t r))
```

Having the function `lam-rec` which does recursion and not just iteration it is easy to inspect an encoded lambda term, and see if it is a variable, an application or an abstraction and return the arguments.

```
is-var t
    -- Check if 't' is a variable. If yes, return 'some j'
    -- where 'j' is the De Bruijn index of the variable.
    -- If no, return 'none'
:=
    lam-rec t zero
        (\ j b        := some j)
        (\ t tr u ur  := none)
        (\ t tr       := none)


is-app t
    -- Check if 't' is an application. If yes, return 'some p'
    -- where 'p' is the pair consisting of the function term
    -- and the argument term. If no, return 'none'
:=
    lam-rec t zero
        (\ j b := none)
        (\ u ur v vr := some (pair u v))
        (\ t tr := none)
```

```
is-lam t
    -- Check if 't' is a lambda abstraction. If yes,
    -- return 'some bdy', otherwise return 'none'.
:=
    lam-rec t zero
        (\ j b       := none)
        (\ f fr a ar := none)
        (\ bdy bdyr  := some body)
```

Note that in the functions `is-app` and `is-lam` it is important to have access to the subterms and not only to the results of the subterms. The results of the subterms are thrown away. In order to check if a term is an application it is not necessary to know if the subterms are applications.

6.4 Redex Reduction

In order to do reduce a redex we just have to transcribe the recursive definition of the shift operator $\uparrow_b^n t$ and the substitution $e[a/i]$ defined in the chapter 5.1 into the encoding defined in this chapter.

```
shift n b t
    -- Shift all De Bruijn indices in the term 't' up by 'n'
    -- starting at the cutoff index 'b'.
:=
    t b v app lam where
        v i b := (i < b) (var i) (var (i + n))
        -- 'app' and 'lam' are just the constructors
        -- for application and lambda abstraction.

subst t i a
    -- Replace in the term 't' the De Bruijn index 'i' by
    -- the term 'a'.
:=
    t i v app lam where
        v j b :=
            lt-eq-gt j b
                (var j)             -- case j < b
                (shift b zero a)    -- case j = b
                (pred j)            -- case j > b
```

6.5 Leftmost Reduction Step

A leftmost reduction step $t \overset{\beta_{\text{lm}}}{\to} u$ is defined formally in 2.10.

The function call `reduce-leftmost t` where t is an encoding of a lambda term shall find the leftmost (sometimes also call leftmost-outermost) redex in the term and reduce it. The function shall return the optional reduced term. The result is an optional term because the term t might already be in normal form.

The following cases have to be distinguished:

1. A variable is in normal form. No leftmost reduction is possible.
2. The term is an application $t_1 t_2$:
   (a) The application has the form $(\lambda t_1) t_2$: The application reduces leftmost to $t_1[t_2/0]$.
   (b) Otherwise:
      i. $t_1 \overset{\beta_{\text{lm}}}{\to} t_{1r}$: $t_1 t_2 \overset{\beta_{\text{lm}}}{\to} t_{1r} t_2$
      ii. t_1 in normal form and $t_2 \overset{\beta_{\text{lm}}}{\to} t_{2r}$: $t_1 t_2 \overset{\beta_{\text{lm}}}{\to} t_1 t_{2r}$
      iii. t_1 and t_2 are in normal form: $t_1 t_2$ is in normal form as well. Note that t_1 cannot be an abstraction. This case has already been handled above.
3. The term is an abstraction. If $t \overset{\beta_{\text{lm}}}{\to} t_r$ then $\lambda t \overset{\beta_{\text{lm}}}{\to} \lambda t_r$. Otherwise the abstraction is in normal form.

```
reduce-leftmost t
    -- Reduce the leftmost redex in 't' and return 'some tr'
    -- if there is one and 'none' if the term is in normal
    -- form.
:=
    lam-rec zero v a l where
        v j b :=
            None

        a t1 r1 t2 r2 :=
            is-lam t1
                (\ bdy :=        -- contract redex
                     subst bdy zero t2)
                (r1
                    (\ t1r := some (app t1r t2))
                    (r2
```

```
                    (\ t2r := some (app t1 t2r))
                    none))

    l t r :=
        r (\ tr := some (lam tr)) none
```

6.6 Normalization

Having the function `reduce-leftmost` which performs a leftmost reduction step and the function `fixpoint` defined in chapter 4 it is trivial to define a function `normalize` which computes the normal form of any lambda term given as an encoding of the lambda term.

```
normalize t
    -- Compute the normal form of the encoded lambda
    -- term 't' by applying a leftmost reduction strategy.
    -- If the term 't' does not reduce to a normal form,
    -- the function does not terminate.
:=
    fixpoint reduce-leftmost t
```

References

[1] H. Brandl, "Lambda Calculus - Step by Step," *WWW*, 2019. [Online]. Available: https://hbr.github.io/Lambda-Calculus/lambda1/untyped_lambda.html

[2] ——, "Programming in lambda calculus," *WWW*, 2020. [Online]. Available: https://hbr.github.io/Lambda-Calculus/lambda2/lambda.html

[3] H. P. Barendregt, *The Lambda-Calculus, its syntax and semantics*, ser. Studies in Logic and the Foundation of Mathematics. North Holland, 1984, second edition.

[4] H. Brandl, "Typed Lambda Calculus / Calculus of Constructions," *WWW*, 2022. [Online]. Available: https://hbr.github.io/Lambda-Calculus/cc-tex/index.html