Programming with Lambda Calculus

Helmut Brandl

Abstract

An introduction into lambda calculus emphasizing the use of lambda calculus as a programming language.

Contents

Motivation

Lambda calculus is a fascinating topic for the following reasons.

1. It is simple. It just consists of variables, functions and function applications.

2. Despite of being simple it is possible to express any computable function in the calculus.

   This is the reason why Alonzo Church (1936) invented the lambda calculus. He wanted to explore the limits of computability and decidability. If you want to prove that something is undecidable you need a clear definition what you mean by *computable* or *decidable*. Lambda calculus is the proper tool.

3. It is fun. In lambda calculus we don't care about execution. We just express functions. And as you will see: We can express arbitrarily complex and interesting functions. No matter if their execution would last longer than the universe exists.

4. We can learn a lot about computation. In lambda calculus we can express iteration, recursion, arbitrary data structures by using only variables, functions and function applications.

In the following I do not expect any prior knowledge. The reader should have some experience with programming.

All concepts are introduced step by step.

Many texts on lambda calculus use a lot of math. The goal usually is not to do programming in lambda calculus, only to demonstrate its computational power.

In this text we use lambda calculus as a programming language. We build first simple functions and step by step compose the simple functions to more complex functions. At the end, the same goal is achieved: Demonstrate the expressive power of lambda calculus. But I hope that leaving out math notation makes the topic more accessible.

Comments, Questions, Remarks, Discussion

Feel free to post any question, comment, remark or discusion as a github issue on https://github.com/hbr/Lambda-Calculus/issues

Basics of Lambda Calculus

Definitions

Terms and Reduction

In lambda calculus everything is a function. A *lambda term* is either a variable, a function application or a function abstraction.

```
term ::=    x              -- variable
        |   \ x := term    -- function
        |   term term      -- application
```

We pronounce the term `\ x := e` *lambda x body e*. The backslash should remind us of the greek letter λ.

The most interesting term is the function term.

```
\ x := term
  ^    ^-- body (might contain variable x)
  |
  \------  bound variable
```

The bound variable is meaningful only within the body. We can change the name of the bound variable arbitrarily as long as we change it consistently within the body.

What can we do with a function? We can apply it to an argument and get `(\ x := e) a`. This term is called a reducible expression (short redex). As the name says, this expression can be reduced or we can compute the result of the function applied to the argument.

```
(\x := e) a ~> e[x:=a]
```

where `~>` reads *reduces to* and `e[x:=a]` is the expression `e` where all occurrences of the variable `x` have been replaced by `a`. A reduction is the most elementary *computation step* in lambda calculus.

We pronounce the substitution `e[x:=a]` *e with a for x*.

Example: Application of the identity function

```
(\x := x) a ~> x[x:=a] = a
```

Conventions

There are some conventions which make writing lambda terms more convenient.

Function application associates to the left

```
f a b c              =           ((f a) b) c
```

Nested function abstractions associate to the right

```
\ x := \y := e       =           \x := (\y := e)
```

and the arguments of nested function abstractions can be compressed.

```
\ x := \y := e       =           \ x y := e
```

Rename Bound Variables

The names of bound variable are irrelevant. We can change the names arbitrarily without changing really the term.

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

```
\ x y := x          =          \ z u := z

\ x y := y          =          \ z u := u

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

This is the same as with any other programming language. The names of formal arguments of a function are irrelevant to the caller of the function.

Furthermore we have to make sure to choose names for bound variables which cannot change the meaning of an expression.

The last example demonstrates that a variable is always bound by the innermost binder.

Substitution

Since the basic computation step is based on substitution, we better define substitution exactly.

```
x[x:=e]             =       e                   -- same variable

y[x:=e]             =       y                   -- different variables

(a b)[x:=e]         =       a[x:=e]  b[x:=e]    -- independent substitution

(\y := t)[x:=e]     =       \y := t[x:=e]       -- pull into abstraction
        -- y must not occur in e !!              -- hygiene condition
```

The effect of the substitution applied to a variable depends on the variable name.

The substitution of an application is done on both terms independently.

We are allowed to pull a substitution into a lambda abstraction only if the replacement term does not contain the bound variable of the abstraction.

This is a *hygiene condition* which is not really a restriction, because we can always rename the bound variable so that the hygiene condition is satisfied.

The hygiene condition guarantees that the name of the bound variable does not *interfere* with variable names in the world outside the abstraction.

Substitution Example

```
(\ x y := x) a b

=   (\x := (\ y := x)) a b

~>  (\ y := x)[x:=a] b          -- y not in a !!

=   (\ y := a) b
```

```
~>  a[y:=b]

=   a
```

The hygiene condition tells us that `y` must not occur in `a` (otherwise we would have to rename `y`).

So we get

```
(\ x y := x) a b    ~>      a
```

i.e. the term `\x y := x` applied to two arguments returns the first argument and ignores the second argument.

In the same manner

```
(\ x y := y) a b    ~>      b
```

can be shown.

Combinators

Lambda terms where all variables are bound are called *combinators*.

Projections

We have seen already the combinators `\ x y := x` and `\x y := y`. In order to use them later on we give them names.

```
K   :=  \ x y := x
KI  :=  \ x y := y
```

We have chosen the names `K` and `KI` to be inline with the literature on lambda calculus which uses the names *K combinator* and *KI combinator*.

These two definitions give just names to the coresponding lambda terms. The names are only for humans. The lambda calculus does not know of any names which we give to combinators. I.e. whenever we see `K` in a lambda term we always mean the term `\ x y := x`.

In order to write the definition of `K` and `KI` more like function definitions in a programming language we use the syntax

```
K x y := x

KI x y := y
```

which we pronounce *K x y with body x*. We just replaced the backslash by a name. Therefore we call `\ x y := x` an anonymous or unnamed function and `K x y := x` a named function.

But note that we always mean the same thing. `K x y := x` is the same as `K := \ x y := x`. The first form just uses syntactic sugar.

Partial Application and Storage

What happens if we apply a function with two arguments to only one argument?

Nothing special. Lambda calculus only knows of functions with one argument. However the function can return another function which then can be applied to the remaining argument.

But let's see what happens of we apply the `K` combinator to one argument.

```
K a

=   (\ x y := x) a

=   (\ x := (\ y := x)) a

~>  (\ y := x)[x:=a]

=   \y := a                  -- y must not occur in a!!
```

Now we have a function which ignores any arguments to which it is applied. The term `K a` *stores* the term `a` and returns it to any other term which applies `K a` to an arbitrary argument.

Note that the hygiene condition is important to guarantee that `K a` returns exactly `a` if applied to another argument. If we hadn't required that `y` must not occur in `a`, then the application `K a b` would return `a[y:=b]` which in that case could be different from `a`.

Boolean

Lambda calculus does not have any primitive values. It only has functions — nothing else.

We have to find a way to express boolean values as functions.

Since there are only two boolean values, we can encode booleans as functions taking two arguments. The boolean value *true* returns the first argument and the boolean value *false* returns the second argument.

I.e. booleans are *decisions* which decide between two alternatives where the alternatives are represented the by the two function arguments.

```
true x y  := x

false x y := y
```

Note that `true` and `K` and `false` and `KI` are defined by the same lambda term. We use different names to express different intentions. But again: This is just for us. From the viewpoint of the lambda calculus the terms are equivalent.

It is not too difficult to define boolean functions. Negation can be defined as

```
not b := b false true
```

The correctness of the definition can be shown by the following reductions.

```
not true

~>  true false true         -- definition 'not'

~>  false                   -- 'true' selects the first argument


not false

~>  false false true        -- definition 'not'

~>  true                    -- 'false' selects the second argument
```

More boolean functions

```
and a b :=  a b false   -- if 'a' is true, return 'b', otherwise 'false'

or a b  :=  a true b    -- if 'a' is true, return 'true', otherwise 'b'
```

Pair

A lambda term representing a pair of values has to *store* in some sense the values. We have already seen that lambda calculus is able to store one value. Remember the term `K a` which *stores* the value `a` and returns it if another argument is given to the term.

The right choice to represent a pair is a lambda term which expects three arguments. The first two arguments are the terms which form the pair of values. The third argument is a function using these two values as arguments.

```
(,) x y := \ f := f x y
```

Here we use the comma operator to use the notation `(a,b)` to express the pair of `a` and `b`. We could have chosen e.g. the name `pair` and written `pair a b` instead of `(a,b)`.

The term `(a,b)` is a partial application. It expects a further argument which should be a function taking two arguments.

We can use the combinators `K` and `KI` to extract either the first or the second component of the pair.

```
first p := p K

second p := p KI
```

Let's see this in action on the term `first (a,b)`.

```
first (a,b)

=   (a,b) K                     -- definition of 'first'

=   (\ x y f := f x y) a b K    -- definition of '(,)'

~>  K a b

~>  a
```

Type Annotations

Untyped lambda calculus does not know of any types. Therefore the name. However the programmer thinks in types. We have already talked about booleans and pairs. These are *types*. We use types to express our intentions.

Since we want to do *programming* in lambda calculus, we want to be able to express our intentions in the source code.

Example: Boolean

We have already seen, that a boolean value is represented as a choice between the two arguments which follow. Usually our intention is to express a choice between two things of the same kind, and not a choice between a string and a number.

The lambda calculus does not care what arguments you provide. But we can use types to express our intentions.

```
true (x: A) (y: A): A   := x

false (x: A) (y: A): A  := y
```

We use capital letters to express any type. Here the definitions tell us, that we can use the functions `true` and `false` on any two arguments provided that they have the same *type*. Both functions then return a value of exactly that type.

But what type do the terms `true` and `false` have? They have the type

```
true:  A -> A -> A
false: A -> A -> A
```

We can use `Boolean` as an abbreviation for `A -> A -> A`.

Since these are just annotations which are ignored by the compiler, we don't need formal rules. Otherwise we would have to switch to *typed lambda calculus* which we won't do here.

Remember: **Type annotations are comments**.

Example: Pair

If `(a,b)` is a pair of values, then the types of `a` and `b` can be different. However the type of the third argument, the function which uses the two values has to be consistent with the component types.

We can express the constraint in the following way.

```
(,) (x: A) (y: B): (A -> B -> C) -> C
:=
    \ f := f x y
```

The type of `(,)` is then

```
(,): A -> B -> (A -> B -> C) -> C
```

Since we are not formal on type annotations we can use `Pair A B` to express the type of a pair where the first component is of type `A` and the second component is of type `B`.

We can add type annotations to the terms `K`, `KI`, `first` and `second` as well.

```
K (x: A) (y: B): A  := x

KI (x: A) (y: B): B := y

first (p: Pair A B): A := p K

second (p: Pair A B): B := p KI
```

If we use type annotations, then we have to type a little bit more. However the definitions are much more readable. Understanding the definitions in 3 months from now is easier with type annotations.

And our version of lambda calculus looks more and more like a programming language.

Terminating and Non Terminating Computations

Up to now all lambda terms we have used have *terminated* in the sense that we reached a state, where no more reducible expressions are in the term.

Is this always the case? Unfortunately not. In the following we show how to construct potentially endless loops in lambda calculus.

Normal Forms

We call a lambda term to be in normal form if it contains no more redexes.

A term not in normal form can contain one or more redexes. A computation step is done by choosing any redex and reduce it. Therefore evaluating a lambda term might be non deterministic.

A reduction sequence might lead to a term in normal form or be an infinite sequence.

```
t1 ~> t2 ~> t3 ~> .......  ~> tN        -- 'tN' is in normal form

t1 ~> t2 ~> t2 ~> ............................
```

We call a lambda term strongly normalizing, if any sequence of reduction steps finally leads to a term in normal form.

We call a lambda term weakly normalizing, if there is a reduction sequence which leads to a term in normal form.

Loops

We can apply a lambda term to itself. This is the easiest form to build an infinite loop.

```
M x := x x
```

Let's see what happens if we apply `M` to itself.

```
M M

=   (\ x := x x) M

~>  (x x)[x:=M]

~>  M M

~>  M M

...
```

I.e. the term `M M` is neither strongly nor weakly normalizing. It is diverging.

The term `M M` is not useful at all. It is just a silly term to demonstrate what can go wrong, because an infinite loop is certainly not desirable.

But we can construct another term which is only potentially non terminating.

```
U x f := f (x x f)
```

Let's see what happens if we apply `U U` to an arbitrary argument.

```
U U g

=   (\ x f := f (x x f)) U g

~>  g (U U g)

~>  g (g (U U g))

~>  g (g (g (U U g)))

...
```

Here we get a potentially infinite reduction sequence. But we can choose `g` in a manner that it might in some states ignore its argument i.e. which might generate some *exit condition* from the loop.

In this introductory chapter we don't look deeper into this possibility. We just wanted to demonstrate that we can program *loops* in lambda calculus which are potentially infinite.

Arithmetic

Church Numerals

We want to encode the natural numbers `0, 1, 2, ...` in lambda calculus. Since lambda calculus only has functions, we have to figure out a way to encode numbers as functions.

What can we do with a number `n`? We can do something `n` times. And what can lambda calculus do? Correct answer: **Function application**.

Therefore we encode a natural number as a function which takes a function argument and a start value and iterates the function `n` times on the start value.

This is called the *Church encoding* of numbers and the encoded numbers are called *Church numerals*.

So we encode the number zero as

```
zero f s := s
```

The successor function just takes a church numeral and applies the function one more time.

```
successor n :=
    \ f s := f (n f s)
```

We can use the combinators `zero` and `successor` to generate arbitrary church numerals.

```
one := successor zero

two := successor one

...
```

Let's check, if the definition really behaves as expected.

```
one

=   successor zero              -- definition 'one'

=   \ f s := f (zero f s)       -- definition 'successor'

~>  \ f s := f s                -- apply 'zero' to 'f' and 's'
```

I.e. we see that `one` really applies the function `f` once on the start value `s`.

Let's do the same for `two`.

```
two

=   successor one               -- definition 'two'

=   \ f s := f (one f s)        -- definition 'successor'

~>  \ f s := f (f s)            -- see previous derivation
```

So we see the function `f` applied 2 times on the start value `s`.

Simple Arithmetics

Church numerals are iterations. We can use that to do simple arithmetics. To add the numbers `n` and `m` which are represented as church numerals we simply apply the successor function `n` times with start value `m`.

```
(+) n m :=
    n successor m
```

Multiplication of the numbers `n` and `m` is defined as `n` times the iterated addition of `m` on the number `zero`.

```
(*) n m :=
    n ((+) m) zero
```

The exponentiation `n ^ m` is defined as `n * n * .... * n * one`, i.e. it is an `m` times iterated multiplication. There is no problem to define exponentiation in lambda calculus

```
(^) n m :=
    m ((*) n) one
```

Simple Predicates

A predicate is a function returing a boolean value. Predicates are *deciders*. We want to be able to decide, if a number is zero, is an even number or is an odd number.

The encoding of the predicate `isZero` as an iteration is surprisingly simple. Evidently the start value of the iteration is `true`, because the number `zero` is zero. The iteration function just ignores the result of the previous iterations and returns `false`.

```
isZero n :=
    n (\ _ := false) true
```

The evenness and oddness predicates can be represented as iterations as well. The start value for `isEven` is `true` and the start value for `isOdd` is false to return the correct value for the number zero.

On each iteration step we toggle the truth value of the result by using the function `not`.

```
isEven n :=
    n not true

isOdd n :=
    n not false
```

Recursion

Up to now all functions on church numerals have used iterations. This technique has its limits. Assume you want to write the predecessor function which returns zero for zero (since zero has no predecessor) and the actual predecessor for any other number. Trying iteration our function looks like

```
predecessor n :=
    n (\ x := ?) zero
```

The start value is clear. According to the definition it has to be `zero`. But how to design the step function? The task of the step function is to map the predecessor of the predecessor into the predecessor of the current number. In nearly all cases adding one does the job. But it fails for the number one, since the predecesssor of `zero` is `zero` we would compute `one` as the predecessor of `one` which is wrong.

Alonzo Church, the inventor of the lambda calculus, had been puzzled to find a proper definition of the predecessor function. A difficult situation when you want to define a calculus where all computable functions can be encoded and the calculus fails on such a simple task as to compute the predecessor of a natural number.

One of his phd students, the mathematician Stephen Kleene (pronounced Klaynee), came up with a solution which not only let us encode the predecessor function, but also a lot of other complex functions.

The problem with iteration: The step function has only access to the function result of the function called on the predecessor argument, but not to the value of the predecessor itself. The iteration *consumes* the numbers. This is clear if we look at the type signatures of the start value `s` and the iteration function `f`.

```
s: A
```

```
f: A -> A
```

`s` has a value of some type `A` and the iteration function `f` maps step by step the value into its final result value. What we want is the following types:

```
s: A
```

```
f: Natural -> A -> A
```

We want the step function `f` having access to an iteration counter and the result of the overall function for this iteration counter. The function `f` then computes the result of the next iteration.

We can reach this goal if we do the iteration with the pair ‘(iteration counter, result) and finally extract the second component of the pair.

For the predecessor function trying to compute the predecessor of a number n we would expect the following sequence of pairs

```
(0, 0)
(1, 0)
(2, 1)
....
(n, n-1)
```

or more generally

```
(0, result for iteration 0)
(1, result for iteration 1)
(2, result for iteration 2)
....
(n, result for iteration n)
```

where at the end we extract the second component of the pair.

The following function does exactly that

```
nat-rec (n: Natural)
        (f: Natural -> A -> A)
        (s: A):
        A
```

```
:=
    second (n step (zero, s)) where
        step p :=
            p (\ i res :=
                    (i + one, f i res))
```

Note how the `step` function has two components. The first component just increments the iteration counter and the second component uses the function `f` to compute from the iteration counter and the result of the previous iteration the result of the current iteration.

Having such a generic recursor, the encoding of the predecessor function is just a piece of cake.

```
predecessor n :=
    nat-rec n (\ i _ := i) zero
```

`predezessor zero` is `zero` because of the start value `zero`. `predecessor one` is `zero` because the iteration counter is `zero`. `predecessor n` receives at the last iteration the iteration counter `n - 1` which is exactly the required result. The iteration function `f` just used the iteration counter to compute the next result and ignores the result of the previous iteration.

Based on the function `predecessor` we can encode the difference between two natural numbers

```
(-) n m :=
    m predeccessor n
```

Coding of comparison functions is now easy.

```
(<=) n m :=
    isZero (n - m)

(<) n m :=
    successor n <= m

equal n m :=
    n <= m and m <= n
```

Even the factorial function computing $1 \times 2 \times 3 \times ... \times n$ is now possible which uses the recursive definition

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \quad \text{for } 0 < n \end{aligned}$$

```
factorial n :=
    nat-rec
        n
        (\ i res := res * (i + one))
        one
```

It might not be immediately obvious that `factorial` really does the right thing. In order to be sure, the following table shows the computation steps for different arguments.

argument	computation
0	1
1	1 * 1
2	1 * 1 * 2
3	1 * 1 * 2 * 3

Searching with Predicates

Bounded Search

It is a standard task to find a number `i` which satisfies a certain predicate `p`. We want to write a function which finds the smallest number below a certain bound which satisfies the predicate. In case that no number below the bound satisfies the number, the function should return the bound.

The result cannot be computed by simple iteration, we need the recursor `nat-rec`. The recursor gives to the step function always the iteration counter `i` of the previous step and the result `res` of the previous step.

We want the recursor to maintain the invariant that all numbers below the previous result do not satisfy the predicate i.e. to maintain

```
for all j: j < res => not (p j)
```

It is easy to find a start value for `res` which satisfies the invariant. Just use `zero`, because there is no number below 0 and therefore all numbers below 0 do not satisfy `p`.

As long as `res` does not satisfy `p`, we increment `res` by one.

As soon as we encounter the first value of `res` which satisfies `p res` we have encountered the smallest number and therefore we do not change the value of `res` anymore.

With this preparation, it is easy to write the function `least-below` and convince ourself that the implementation is correct.

```
least-below (n: Natural) (p: Natural -> Boolean): Natural :=
        -- Least number 'i' below 'n' satisfying 'p i'
        -- or 'n' if there is no such number.
    n (\ res := p res res (res + one)) zero
        -- maintain the invariant: all numbers below
        -- 'res' do not satisfy 'p'.
```

The expression `p res` checks if `res` satisfies the predicate. If the answer is `true`, then the value `res` is kept for the next iteration. If the answer is `false`, then

the value of `res` is incremented. If no number below `n` satisfies the predicate `p`, then the value `zero` is incremented `n` times, i.e. `n` is returned as the final result.

Next we want to implement an existential quantifier with an upper bound.

If we find a number below the bound which satisfies a certain predicate, we know that at least one number below the bound exists, which satisfies the predicate. The existential quantifier with an upper bound can be implemented by looking at the result of `least-below` and comparing it with the bound.

```
exist-below (n: Natural) (p: Natural -> Boolean): Boolean :=
        -- Is there a number 'i' below 'n' satisfying 'p i'?
    least-below n p < n
```

If there exists no number below a bound which does not satisfy a certain predicate, then all numbers below the bound satisfy the predicate. Therefore implementation of the universal quantifier with an upper bound is easy as well.

```
all-below (n: Natural) (p: Natural -> Boolean): Boolean :=
        -- Are all numbers 'i' below 'n' satsifying 'p i'?
    not
        (exist-below
            n
            (\ x := not (p x)))
```

Division

With these helper functions based on predicates we can implement division functions and functions computing prime numbers.

The value of `a` divided by `b` i.e. `a / b` is the unique solution `x` of the inequalities

```
b * x <= a
```

```
a < b * (x + 1)
```

Division by zero is undefined. In the case `b = 0` the second inequality is cannot be satisfied. In that case we want the expression `a / b` to return `a` in order to have a total function.

We can use `least-below` with upper bound `a` to find the smallest number `x` which satisfies the second inequality. In case that no such numbers exist, we get as expected the upper bound `a`. But are we sure that the first inequality is satisfied?

From the reasoning above we know, that the function `least-below` maintains the invariant for all numbers strictly below `x`

```
not (a < b * (x + 1))
```

which is equivalent to

```
b * (x + 1) <= a
```

and

```
b * x + b <= a
```

which in turn implies

```
b * x <= a
```

Therefore the first inequality is maintained by the function `least-below`.

I.e. the following implementation is correct.

```
(/) a b :=
    least-below
        a
        (\ x :=
            a < b * (x + one))
```

The function `divides a b` shall decide, if `a` divides `b` exactly i.e. if there exist a solution `x` satisfying

```
x * a = b
```

The number `b / a` is a good candidate for the solution `x`, because according to its definition it satisfies

```
b / a * a <= b
```

So we just compute `b / a * a` and compare it with `b`.

```
divides a b :=
    equal (b / a * a) b
```

Prime Numbers

Prime numbers are very important in number theory and cryptography. In this section we show the implementation of some important prime number functions.

A prime number is a natural number greater than 1 which is only divisible by 1 and itself.

If we reformulate the definition a little bit, we can implement it and get a prime number tester in lambda calculus.

```
isPrime n :=
    one < n
    and
    all-below
        n
        (\ x :=
            x <= one
            or
```

```
            not (divides x n))
```

If we want to compute the `n`th prime number, we have to think a little bit.

We know that `two` is the first prime number.

If we have the `i`th prime number `pi`, we get the next prime number by finding the smallest number `x` strictly above `pi` which satisfies `isPrime x`. In order to use the function `least-below` we need an upper bound for the search.

Let's find an upper bound for the next prime number above `pi`. We form the product `z = p0 * p1 * ... * pi` of all prime numbers below `pi` including `pi`. Certainly none of these prime numbers divides `z + 1` because each division leaves the remainder 1. Therefore `z + 1` is either a prime number or there exists a prime number different from the prime numbers in the product dividing `z + 1`. Therefore `z + 2` is a strict upper bound for the next prime number.

Next we observe that `z <= factorial pi` is valid, because the factorial is the product of more numbers than `z`. Therefore `factorial pi + 2` is a strict upper bound for the next prime number above `pi`.

Now the implementation is straightforward.

```
nth-prime n :=
    n f two where
        f p_i :=
                -- p_i is the 'i'th prime
            least-below
                (factorial p_i + two)
                (\ x := p_i < x and isPrime x)
```

> Note that the above reasoning to find an upper bound for the next prime number is the reasoning which has been used by Euclid to prove that there are infinitely many prime numbers.

From number theory we know that every natural number above zero has a unique prime number factorisation. I.e. each positive number `n` can be written as the infinite product

```
n = p0 ^ e0 * p1 ^ e1 * p2 ^ e2 * ...
```

where `pi` is the `i`th prime number and `ei` is the corresponding exponent. For `n = 1` all exponents are 0.

We want to have a function which computes for all numbers `n` the exponent `ei` of the `i`th prime number.

For each pair `(pi,ei)`

```
divides (pi ^ k) n
```

is valid for all `k <= ei` and

```
divides (pi ^ (ei + 1)) n
```

is invalid.

Therefore we can find the exponent by a search for the least number which does not satisfy the last proposition.

The upper bound for the search is easy to find. Since all prime numbers are greater than 1, all exponents are lower than `n`.

```
prime-exponent i n :=
        -- Exponent of the 'i'th prime number in 'n'
    least-below
        n
        (\ x :=
            not (divides
                    ((nth-prime i) ^ (x + one))
                    n))
```

The prime factorization for the number 0 is not defined. However the function `prime-exponent i zero` returns `zero`. This is no problem. We have just assigned an arbitrary result to the function for arguments, where it is mathematically undefined. For all other arguments the function returns the correct exponent.

Unbounded Search

If we have a predicate `p: Natural -> Boolean` and know that there exists a number which satifies the predicate, then we can find the least number by an unbounded search. In traditional programming languages we would use a while-loop which has the continuation condition `not (p i)` and which increments in the body of the loop the number by one until the continuation condition is violated.

In lambda calculus we don't have while loops. Therefore we have to find a way to do the search with functions.

We would like to write the function in the following form

```
search-least (p: Natural -> Boolean): Natural :=
    iterate step zero where
        step := ?
        iterate := ?
```

which iterates a step function as long as needed starting with `zero` and maintaining the invariant, that all numbers below the current number do not satisfy the predicate.

The function `step` needs as an argument the current number to check. We could try the following.

```
step i :=
    p i i ?
```

If the term `p i` returns true, then `p i i ?` returns the value `i` which satisfies the predicate. But we don't know what to return in case that `i` does not satisfy the predicate. If we had recursive functions in lambda calculus we would just replace the question mark with `step (i + one)`. Unfortunately this is not a valid lambda term.

But we can give the step function another argument, which is a continuation (traditionally called `k`) knowing how to do the rest of the computation.

```
step k i :=
    p i i (k (i + one))
```

Now the rest of the difficulty remains on the unkwnown term `iterate`. We just know that this term has to do some kind of *self replication* to implement the loop. In the chapter *Basics of Lambda Caluculus* we have already encountered a combinator `U` which does some kind of replication.

```
U x y := y (x x y)
```

The combinator `U` expects two arguments and returns a term which contains both arguments twice. It is interesting to see what happens, if we evaluate `U U step i`. We get

```
U U step i    ~>  step (U U step) i
```

I.e. `U U step i` calls `step` with `U U step` as first argument and `i` as second argument. Now `step` is in the function position and has control of what to do next. The function `step` evaluates `p i`. If the result is `true` then it returns `i` and the iteration terminates. If `p i` evaluates to `false` then it returns `(U U step) (i + one)` and the iteration can continue.

The iteration is started with `U U step zero` i.e. we can use `U U` as `iterate` and we are ready.

We see, that we have implemented an iteration which stops, as soon as a number is encountered which satisfies the predicate. The complete function reads

```
search-least (p: Natural -> Boolean): Natural :=
    U U step zero where
        step k i :=
                -- invariant: all numbers below `i` do not
                -- satisfy `p i`.
            p i i (k (i + one)
        U x y :=
            y (x x y)
```

It might be necessary to read this section twice or more to understand the tricky mechanism to implement the unbounded search. But it is possible.

However I admire the genious, who invented it. I would have never found such a cleverly constructed lambda term by myself.

Some remarks:

- All functions constructed before this section on unbounded search are strongly normalizing (provided that their arguments are of the proper kind). The function `search-least` is only weakly normalizing (provided that a number exists which satisfies the predicate, otherwise it is diverging).

  I.e. there are only some reduction sequences, which terminate with the desired result. But there are other reductions sequences which are infinite. The subterm `U U step` has an infinite reduction sequence, because it reduces to `step (U U step)` which contains itself as a subterm.

  However there are reduction strategies, which find for all weakly normalizing terms a reduction path which terminates.

- As long as you remain in constructive mathematics, you don't need unbounded search. Unbounded search needs a guarantee, that a number satisfying the predicate exists. In constructive mathematics an existence proof requires a construction of an object which satisfies the condition. But if you have a construction of such an object, you can use it as an upper bound and use `least-below` to find the smallest number satisfying the predicate.

- The availability of unbounded search makes lambda calculus as expressive as general recursive functions. The class of general recursive functions consists of the constant zero, the successor function, all projections (`K` and `KI` cover the special case with two arguments, but the generalization to more argumentes is obvious) and are closed under primitive recursion (`nat-rec`) and minimization (*unbounded search*).

  There are many definitions of computable functions. E.g.

  - Recursive functions
  - Turing machines
  - Lambda calculus

  Fortunately it can be proved that they are all equivalent i.e. they define the same class of functions.

Data Types

Data types are an important means to structure computations. We think of data types like `Boolean`, `Natural`, `List`, `Tree`, etc. In the previous chapters represented the types `Boolean` and `Natural` in lambda calculus.

Since lambda calculus has only functions, we have to represent objects of a certain type as functions. We represented the type `Boolean` as a function taking two arguments and returning one of the them depending on its boolean value.

We represented natural numbers as functions taking two arguments. The first argument is a function and the second a start value. A natural number iterates the function `n` times on the start value.

It is possible to define any datatype in lambda calculus. In textbooks on lambda calculus you are many times shown that it is possible e.g. to represent lists. It sometimes look like some rabbit has been pulled out of the hat by some magic.

But no magic and no genious is needed to find lambda representations of datatypes. There is a construction principle which is fairly general. In this chapter we present the construction principle and show it on some old (natural numbers) and new (lists and trees) examples.

Construction Principle

In order to understand the construction principle, we have to understand a seemingly unrelated topic: Algebra.

An important example of an algebra is the concept of a *group*. A group must have

- a set of elements
- a unit element (i.e. a constant)
- a unary function which applied to an element returns its inverse
- a binary function which let us combine elements

Apart from that it takes some properties. The unit element must be neutral with respect to the binary operation. An element combined with its inverse is the unit element. The binary operation is associative. However we don't need the properties here. The operations are sufficient.

In a functional language you could define a datatype like

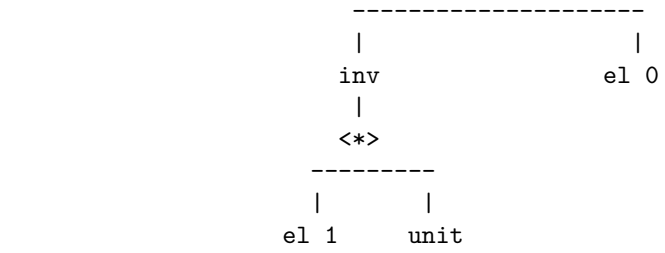
```
class Group A :=
    el:    A -> Group A
    unit:  Group A
    inv:   Group A -> Group A
    (<*>): Group A -> Group A -> Group A
```

with three constructors for the constant, the unary and the binary operation, and one constructor (`el`) to produce elements from some generator set `A`.

We can use such a type to form expressions and each expression has some tree associated with it.

```
-- expression
    inv (el 1 <*> unit) <*> el 0

-- expression tree
                  <*>
                   |
```



The numbers are just a silly example for an arbitrary generator set. Usually the set of generators is finite.

But anyhow. The possible expressions don't have any intrinsic meaning. Speaking in terms of abstract algebra we have defined a *signature*. A *signature* is a set of operation symbols where each operation symbol has an arity. An operation symbol with arity 0 is a constant.

The signature just defines a collection of wellformed expressions in that algebra.

In order to give a meaning (i.e. semantics) to the expressions, we have to define a function which *evaluates* the expression to a value of some type `R` (standing for the result type).

This can be done by assigning a meaning to each operator symbol. For programming this means that every symbol has to get type signature.

symbol	type signature
`el`	`A -> R`
`unit`	`R`
`inv`	`R -> R`
`<*>`	`R -> R -> R`

Note that `Group A` has been replaced by `R` i.e. each group expression corrensponds to some value in its interpretation.

A lambda term representing some group expression is a function with four arguments, one for each operation symbol. Each argument is a function (or constant) according to the type signature. I.e. a lambda term representing a group expression is an **evaluator** of that group expression.

It is straightforward to define the four constructors in lambda calculus.

```
el (a: A): Group A :=
    \ e u i m := e a

unit: Group A :=
    \ e u i m :=
        u
```

```
inv (g: Group A): Group A :=
    \ e u i m :=
        i (g u i m)

(<*>) (g1 g2: Group A): Group A :=
    \ e u i m :=
        m (g1 e u i m) (g2 e u i m)
```

Having this we can write the expression `inv (el one <*> unit) <*> zero` in lambda calculus.

A lambda expression of type `Group A` has the complicated looking type

```
(A -> R) -> R -> (R -> R) -> (R -> R -> R) -> R
```

But remember that we use type annotations as comments to document our intentions. The lambda calculus described here is untyped.

Church Numerals Revisited

Let's revisit church numerals to see how they fit into the construction principle.

In a functional programming language we would define natural numbers as

```
class Natural :=
    successor: Natural -> Natural
    zero:      Natural
```

Interpreted as an abstract algebra:

```
-- expression
    successor (successor zero)

-- expression tree
        successor
           |
        successor
           |
          zero
```

symbol	type signature
`successor`	`R -> R`
`zero`	`R`

A lambda term representing an expression of the algebra must have the type

```
(R -> R) -> R -> R
```

We get the following lambda terms for the constructors

```
zero: Natural :=
    \ f s := s

succ (n: Natural): Natural :=
    \ f s := f (n f s)
```

Lists

Now let's apply the construction pattern to lists and look at a definition of the list type in a functional programming language.

```
class List A :=
    cons: A -> List A -> List A
    nil:  List A
```

We interpret the type definition as the definition of an algebra and look at the expressions it generates and at corresponding expressions trees.

```
-- expression
    cons 0 (cons 1 (cons 2 nil))

-- expression tree
           cons
            |
        ---------
        |       |
        0      cons
                |
             --------
             |      |
             1     cons
                    |
                 -------
                 |     |
                 2    nil
```

The definition has two symbols `cons` and `nil` with the following type signatures.

symbol	type signature
`cons`	`A -> R -> R`
`nil`	`R`

Therefore a lambda term of type `List A` must have the type

```
(A -> R -> R) -> R -> R
```

We apply the construction principle and get the lambda terms for the constructors.

```
nil: List A :=
    \ f s := s

cons (head: A) (tail: List A): List A :=
    \ f s :=
        f head (tail f s)
```

By applying the constructors we can form expressions to construct arbitrary lists e.g. `cons zero (cons one (cons two nil))`.

Each list expression is a lambda term which iterates over the list given the folding function `f` and the start value `s` as arguments.

Some simple list functions:

```
length (list: List A) :=
    list (\ a res := res + one) zero

sum (list: List Natural) :=
    list (\ a res := a + res) zero
```

We get the concatenation of the two lists `a` and `b` by folding `cons` over the list `a` with the start value `b`.

```
concat (a b: List A): List A :=
    a cons b
```

A list reversal is done by reversing the tail and concate the resversed tail with the one element list of the head.

```
reverse (a: List A): List A :=
    a
        (\ head res := concat res (cons head nil))
        nil
```

If we want to compute the tail of a list we face the same problems as with the predecessor function on church numerals. Since the empty list does not have a tail, we accept the empty list as the tail of an empty list.

We can solve the problem in the same manner as with the church numerals. We define a list recursor which internally not only has access to the result of the previous iteration, but also to the previous lists. I.e. the folding function has the type `A -> List A -> R -> R`.

Internally the recursor uses pairs `(tail list, previous result)` starting with `(nil, s)` and throwing away the tail list at the end of the iteration.

```
list-rec (list: List A) (f: A -> List A -> R -> R) (s: R): R
:=
    second (list step start) where
        start :=
            (nil, s)
```

```
        step a p :=
            p (\ tail res :=
                (cons a tail, f a tail res))
```

Now we can define the function `tail`.

```
tail (list: List A): List A :=
    list-rec
        list
        (\ head tail res := tail)
        nil
```

In the same manner we can define a function `head`. In that case we have to provide a default value since we cannot pull out an arbitrary list element from an empty list.

```
head (list: List A) (default: A): A :=
    list-rec
        list
        (\ hd tl res := hd)
        default
```

Trees

I hope that the construction principle becomes clearer and clearer. As a last expample we show how to represent binary trees as lambda terms.

In order to keep things simple we look at binary trees which store information only in the leaves.

A type definition of such a tree in a functional programming language looks like

```
class Tree A :=
    node: Tree A -> Tree A -> Tree A
    leaf: A -> Tree A
```

As before we look at the expressions and expression trees of the corresponding algebra.

```
-- expression
    node (node (leaf 0) (leaf 1)) (leaf 2)

-- expression tree
            node
             |
        --------------
        |            |
       node        leaf 2
        |
    -----------
```

```
    |         |
  leaf 0   leaf 1
```

symbol	type signature
`node`	`R -> R -> R`
`leaf`	`A -> R`

A lambda term representing a tree has two arguments one representing the binary operation on the node and one which maps the leaf information into the result type. I.e. the type `Tree A` is represented by a lambda term having the type

```
(R -> R -> R) -> (A -> R) -> R
```

The lambda terms representing the two constructors look like.

```
leaf a :=
    \ nd lf :=
        lf a

node a b
    \ nd lf :=
        nd (a nd lf) (b nd lf)
```

With these constructors we can form tree expressions like `node (node (leaf 0) (leaf 1)) leaf 2`.

Some simple functions on trees:

```
count-nodes tree :=
        -- Count the nodes in a tree including the leaf nodes.
    tree
        (\ size-a size-b := one + size-a + size-b)
        (\ _ := one)

flip tree :=
        -- Make a mirror image of the tree.
    tree
        (\ left right := node right left)
        leaf


to-list (tree: Tree A): List A :=
        -- Transform the tree into a list of leaf values
    tree
        concat
        (\ el := cons el nil)
```

Exercise: Define a binary tree in lambda calculus where all information is stored in the nodes and the leaves are empty. Implement the functions `count-nodes`, `flip` and `to-list` for this kind of tree.

I hope that by looking at these examples it should be easy to represent any data type which can be represented in a functional language.

This ends this introduction into programming with lambda calculus. I hope you enjoyed it.

References

Church, Alonzo. 1936. "An Unsolvable Problem of Elementary Number Theory." *Journal of Mathematics* 58: 354–63.