9

PROGRAMMING IN UNTYPED λ-CALCULUS

This section explains how to use untyped λ-calculus to encode data structures and useful data, such as booleans, linked lists, natural numbers or binary trees. All this is done on pure λ-calculus avoiding the addition of new syntax or axioms.

This presentation follows the Mikrokosmos tutorial on λ-calculus, which aims to teach how it is possible to program using untyped λ-calculus without discussing technical topics such as those we have discussed on the chapter on untyped λ-calculus. It also follows the exposition on [Sel13] of the usual Church encodings.

All the code on this section is valid Mikrokosmos code.

9.1 BASIC SYNTAX

In the interpreter, λ-abstractions are written with the symbol \, representing a λ. This is a convention used on some functional languages such as Haskell or Agda. Any alphanumeric string can be a variable and can be defined to represent a particular λ-term using the = operator.

As a first example, we define the identity function (`id`), function composition (`compose`) and a constant function on two arguments which always returns the first one untouched (`const`).

```
id = \x.x
compose = \f.\g.\x.f (g x)
const = \x.\y.x
```

Evaluation of terms will be presented as comments to the code,

```
compose id id
-- [1]: λa.a ⇒ id
```

It is important to notice that multiple argument functions are defined as higher one-argument functions that return another functions as arguments. These intermediate functions are also valid λ-terms. For example

```
discard = const id
```

is a function that discards one argument and returns the identity, `id`. This way of defining multiple argument functions is called the **currying** of a function in honor to the american logician Haskell Curry in [CF58]. It is a particular instance of a deeper fact: exponentials are defined by the following adjunction

$$\hom(A \times B, C) \cong \hom(A, \hom(B, C)).$$

9.2 A TECHNIQUE ON INDUCTIVE DATA ENCODING

We will implicitly use a technique on the majority of our data encodings that allows us to write an encoding for any algebraically inductive generated data. This technique is used without comment on [Sel13] and represents the basis of what is called the **Church encoding** of data in λ-calculus.

We start considering the usual inductive representation of a data type with constructors, as we do when representing a syntax with a BNF, for example,

$$\texttt{Nat} ::= \texttt{Zero} \mid \texttt{Succ Nat}.$$

Or, in general

$$\texttt{D} ::= C_1 \mid C_2 \mid C_3 \mid \dots$$

It is not possible to directly encode constructors on λ-calculus. Even if we were able, they would have, in theory, no computational content; the data structure would not be reduced under any λ-term, and we would need at least the ability to pattern match on the constructors to define functions on them. Our λ-calculus would need to be extended with additional syntax for every new data structure.

This technique, instead, defines a data term as a function on multiple arguments representing the missing constructors. In our example, the number 2, which would be written as $\texttt{Succ}(\texttt{Succ}(\texttt{Zero}))$, would be encoded as

$$2 = \lambda s.\ \lambda z.\ s(s(z)).$$

In general, any instance of the data structure `D` would be encoded as a λ-expression depending on all its constuctors

$$\lambda c_1.\ \lambda c_2.\ \lambda c_3.\ \dots\ \lambda c_n.(term).$$

This acts as the definition of an initial algebra over the constructors and lets us to compute over instances of that algebra by instantiating it on particular cases. Particular examples are described on the following sections.

9.3 BOOLEANS

Booleans can be defined as the data generated by a pair of constuctors

$$\texttt{Bool} ::= \texttt{True} \mid \texttt{False}.$$

Consequently, the Church encoding of booleans takes these constructors as arguments and defines

```
true  = \t.\f.t
false = \t.\f.f
```

Note that `true` and `const` are exactly the same term up to α-conversion. The same thing happens with `false` and `alwaysid`. The absence of types prevents us to make any effort to discriminate between these two uses of the same λ-term. Another side-effect of this definition is that our `true` and `false` terms can be interpreted as binary functions choosing between two arguments, that is,

- $\texttt{true}(a, b) = a$,
- $\texttt{false}(a, b) = b$.

We can test this interpretation on the interpreter to get

```
true id const
false id const
--- [1]: id
--- [2]: const
```

This inspires the definition of an `ifelse` combinator as the identity

```
ifelse = \b.b
(ifelse true) id const
(ifelse false) id const
--- [1]: id
--- [2]: const
```

The usual logic gates can be defined profiting from this interpretation of booleans

```
and = \p.\q.p q p
or = \p.\q.p p q
```

```
not = \b.b false true
xor = \a.\b.a (not b) b
implies = \p.\q.or (not p) q

xor true true
and true true
--- [1]: false
--- [2]: true
```

9.4 NATURAL NUMBERS

Our definition of natural numbers is inspired by the Peano natural numbers. We use two constructors

- zero is a natural number, written as Z;
- the successor of a natural number is a natural number, written as S;

and the BNF we defined when discussing how to encode inductive data.

```
0    = \s.\z.z
succ = \n.\s.\z.s (n s z)
```

This definition of `0` is trivial: given a successor function and a zero, return zero. The successor function seems more complex, but it uses the same underlying idea: given a number, a successor and a zero, apply the successor to the interpretation of that number using the same successor and zero.

We can then name some natural numbers as

```
1 = succ 0
2 = succ 1
3 = succ 2
4 = succ 3
5 = succ 4
6 = succ 5
...
```

even if we can not define an infinite number of terms as we might wish. The interpretation the natural number n as a higher order function is a function taking an argument `f` and applying them n times over the second argument.

```
5 not true
4 not true
double = \n.\s.\z.n (compose s s) z
double 3
```

```
--- [1]: false
--- [2]: true
--- [3]: 6
```

Addition $n + m$ applies the successor m times to n; and multiplication nm applies the n-fold application of the successor m times to 0.

```
plus = \m.\n.\s.\z.m s (n s z)
mult = \m.\n.\s.\z.m (n s) z
plus 2 1
mult 2 4
--- [1]: 3
--- [2]: 8
```

9.5 THE PREDECESSOR FUNCTION AND PREDICATES ON NUMBERS

The predecessor function is much more complex than the previous ones. As we can see, it is not trivial how could we compute the predecessor using the limited form of induction that Church numerals allow.

Stephen Kleene, one of the students of Alonzo Church only discovered how to write the predecessor function after thinking about it for a long time (and he only discovered it while a long visit at the dentist's, which is the reason why this definition is often called the ***wisdom tooth trick***, see [Cro75]). We will use a slightly different version of the definition that does not depend on a pair datatype.

We will start defining a *reverse composition* operator, called `rcomp`; and we will study what happens when it is composed to itself; that is

```
rcomp = \f.\g.\h.h (g f)
\f.3 (inc f)
\f.4 (inc f)
\f.5 (inc f)
--- [1]: λa.λb.λc.c (a (a (b a)))
--- [2]: λa.λb.λc.c (a (a (a (b a))))
--- [3]: λa.λb.λc.c (a (a (a (a (b a)))))
```

will allow us now to use the `b` argument to discard the first instance of the `a` argument and return the same number wihtout the last constructor. Thus, our definition of `pred` is

```
pred = \n.\s.\z.(n (inc s) (\x.z) (\x.x))
```

From the definition of `pred`, some predicates on numbers can be defined. The first predicate will be a function distinguishing a successor from a zero. It will be user later to build more complex ones. It is built by appliying a `const false` function `n` times to a true constant. Only if it is applied `0` times, it will return a true value.

```
iszero = \n.(n (const false) true)
iszero 0
iszero 2
--- [1]: true
--- [2]: false
```

From this predicate, we can derive predicates on equality and ordering.

```
leq = \m.\n.(iszero (minus m n))
eq  = \m.\n.(and (leq m n) (leq n m))
```

9.6 LISTS AND TREES

We would need two constructors to represent a list: a `nil` signaling the end of the list and a `cons`, joining an element to the head of the list. An example of list would be

$$\texttt{cons}\ 1\ (\texttt{cons}\ 2\ (\texttt{cons}\ 3\ \texttt{nil})).$$

Our definition takes those two constructors into account

```
nil  = \c.\n.n
cons = \h.\t.\c.\n.(c h (t c n))
```

and the interpretation of a list as a higher-order function is its `fold` function, a function taking a binary operation and an initial element and appliying the operation repeteadly to every element on the list.

$$\texttt{cons}\ 1\ (\texttt{cons}\ 2\ (\texttt{cons}\ 3\ \texttt{nil})) \xrightarrow{fold\ plus\ 0} \texttt{plus}\ 1\ (\texttt{plus}\ 2\ (\texttt{plus}\ 3\ 0)) = 6$$

The `fold` operation and some operations on lists can be defined explicitly as

```
fold = \c.\n.\l.(l c n)
sum  = fold plus 0
prod = fold mult 1
all  = fold and true
any  = fold or false
length = foldr (\h.\t.succ t) 0
```

```
sum (cons 1 (cons 2 (cons 3 nil)))
all (cons true (cons true (cons true nil)))
--- [1]: 6
--- [2]: true
```

The two most commonly used particular cases of fold and frequent examples of the functional programming paradigm are `map` and `filter`.

- The **map** function applies a function `f` to every element on a list.
- The **filter** function removes the elements of the list that do not satisfy a given predicate. It *filters* the list, leaving only elements that satisfy the predicate.

They can be defined as follows.

```
map    = \f.(fold (\h.\t.cons (f h) t) nil)
filter = \p.(foldr (\h.\t.((p h) (cons h t) t)) nil)
```

On `map`, given a `cons h t`, we return a `cons (f h) t`; and given a `nil`, we return a `nil`. On `filter`, we use a boolean to decide at each step whether to return a list with a head or return the tail ignoring the head.

```
mylist = cons 1 (cons 2 (cons 3 nil))
sum (map succ mylist)
length (filter (leq 2) mylist)
--- [1]: 9
--- [2]: 2
```

Lists have been defined using two constructors and **binary trees** will be defined using the same technique. The only difference with lists is that the `cons` constructor is replaced by a `node` constructor, which takes two binary trees as arguments. That is, a binary tree is

- an empty tree; or
- a node, containing a label, a left subtree, and a right subtree.

Defining functions using a fold-like combinator is again very simple due to the chosen representation. We need a variant of the usual function acting on three arguments, the label, the right node and the left node.

```
-- Binary tree definition
node = \x.\l.\r.\f.\n.(f x (l f n) (r f n))
-- Example on natural numbers
mytree    = node 4 (node 2 nil nil) (node 3 nil nil)
triplesum = \a.\b.\c.plus (plus a b) c
mytree triplesum 0
--- [1]: 9
```

9.7 FIXED POINTS

A fixpoint combinator is a term representing a higher-order function that, given any function `f`, solves the equation

$$\texttt{x} = \texttt{f x}$$

for `x`, meaning that, if `fix f` is the fixpoint of `f`, the following sequence of equations holds

$$\texttt{fix}\ f = f(\texttt{fix}\ f) = f(f(\texttt{fix}\ f)) = f(f(f(\texttt{fix}\ f))) = \ldots$$

Such a combinator actually exists; it can be defined and used as

```
fix := (\f.(\x.f (x x)) (\x.f (x x)))
fix (const id)
--- [1]: id
```

Where `:=` defines a function without trying to evaluate it to a normal form; this is useful in cases like the previous one, where the function has no normal form. Examples of its applications are a *factorial* function or a *fibonacci* function, as in

```
fact := fix (\f.\n.iszero n 1 (mult n (f (pred n))))
fib  := fix (\f.\n.iszero n 1 (plus (f (pred n)) (f (pred (pred n)))))
fact 3
fib 3
--- [1]: 6
--- [2]: 5
```

Note the use of `iszero` to stop the recursion.

The `fix` function cannot be evaluated without arguments into a closed form, so we have to delay the evaluation of the expression when we bind it using `!=`. Our evaluation strategy, however, will always find a way to reduce the term if it is possible, as we saw in Corollary 1; even if it has intermediate irreducible terms.

```
fix              -- diverges
true  id fix     -- evaluates to id
false id fix     -- diverges
```

Other examples of the interpreter dealing with non terminating functions include infinite lists as in the following examples, where we take the first term of an infinite list without having to evaluate it completely or compare an infinite number arising as the fix point of the successor function with a finite number.

```
-- Head of an infinite list of zeroes
head = fold const false
head (fix (cons 0))
-- Compare infinity with other numbers
infinity != fix succ
leq infinity 6
---- [1]: 0
---- [2]: false
```

These definitions unfold as

- `fix (cons 0)` = `cons 0 (cons 0 (cons 0 ...))`, an infinite list of zeroes;
- `fix succ` = `succ (succ (succ ...))`, an infinite natural number.