Theory of Programming Languages

NPRG085

Vít Šefl & Tomáš Petříček

Charles University

2026

Currying

In lambda calculus, all functions are unary. We can simulate a function with multiple parameters using multiple abstractions.

$$(\lambda x.(\lambda y.x + y))\ 1\ 2 \rightarrow (\lambda y.1 + y)\ 2 \rightarrow 1 + 2 \rightarrow 3$$

This concept is known as *currying*!

Currying

```
def f(x):
    def g(y):
        return x + y
    return g

# f = lambda x: lambda y: x + y

>>> f(10)(5)
15
>>> a1 = f(1)
>>> a2 = f(2)
>>> list(map(a1, range(5)))
[1, 2, 3, 4, 5]
>>> list(map(a2, range(5)))
[2, 3, 4, 5, 6]
```

Recursion

Lambda calculus has no concept of a definition, so one might expect recursion to be impossible. Let us take a look at a standard factorial function.

$$f := \lambda n.\textbf{if}\ (n = 0)\ 1\ (n \cdot f\ (n - 1))$$

We can take the recursive occurrence of f and turn it into a parameter without changing the meaning:

$$f := (\lambda g n.\textbf{if}\ (n = 0)\ 1\ (n \cdot g\ (n - 1)))\ f$$

Recursion

Notice that our definition now has the form:

$$f := G\ f$$

Recall that any x that satisfies $x = h(x)$ is a fixed point of h. In our case, f is a fixed point of the function G. If we can find these fixed points, we can express any recursion!

Define Y as:

$$Y := \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

Recursion

With some equational reasoning, we can show that:

$$\begin{aligned} Y\ G &= (\lambda x.G\ (x\ x))(\lambda x.G\ (x\ x)) \\ &= G\ ((\lambda x.G\ (x\ x))(\lambda x.G\ (x\ x))) \\ &= G\ (Y\ G) \end{aligned}$$

That is, $Y\ G$ is precisely the fixed point f we were looking for! Indeed, $Y\ G\ n$ computes the factorial of n.

Y is a so-called *fixed-point combinator*.

Recursion

This technique can be used to turn any recursive definition into a non-recursive one plus the fixed-point combinator. The Y combinator is used most often, but there is an infinite number of combinators that behave in the same way.

You might be asking: what about *weird* forms of recursion? How do we express, for example, mutual recursion with a fixed-point combinator that only takes a single function?

Recursion

Consider this toy example:

```
odd  n = if n == 0 then False else even (n - 1)
even n = if n == 0 then True  else odd  (n - 1)
```

We cannot really apply the trick we saw before because it just adds an unused parameter and the function still remains recursive. However, we can add a parameter for *both*:

```
odd'  (o, e) n = if n == 0 then False else e (n - 1)
even' (o, e) n = if n == 0 then True  else o (n - 1)
```

Recursion

Now we can process both functions at once by computing the fixed point of their pair as follows:

```
(odd, even) = y (\fs -> (odd' fs, even' fs))
```

```
ghci> map odd [0..5]
[False,True,False,True,False,True]
```

This can be easily generalized to any number of functions:

```
yMany recs = y (\fs -> map ($ fs) recs)
```

Turing Completeness

The previous result seems to indicate that lambda calculus is computationally powerful. Indeed, unbounded recursion allows us to simulate a Turing machine, which makes lambda calculus Turing complete.

However, how do you simulate a state machine and an infinite tape if all you have are functions? We could of course extend lambda calculus with some new data types, but we will show you that functions are perfectly sufficient!

In fact, we can encode all useful data types using only functions. However, whether we want to do so in practice is a different question.

Encoding Booleans

Let us start with booleans, the simplest possible nontrivial type. The usual way to represent a boolean is to store false as `0x00` and true as `0x01`. However, lambda calculus knows nothing about bytes, let alone hardware.

We need to take a step back and think about booleans as an *abstract data type*. We have two ways of constructing values of this type: `true` and `false`. Once we have a boolean, the usual way to use it is to make a choice between two items. If true, we pick the first item; if false, we pick the second item.

Encoding Booleans

Let us take a look at how booleans are used and try to work backward to reconstruct the representation. Consider:

$$\text{choose } \textbf{true } t\ f = t$$
$$\text{choose } \textbf{false } t\ f = f$$

Notice that this is just stating that:

$$\text{choose } \textbf{true} = \lambda tf.t$$
$$\text{choose } \textbf{false} = \lambda tf.f$$

What if we just take those as our representation? That is, set choose $=$ id.

Encoding Booleans

We have:

$$\textbf{true}\ t\ f = t$$
$$\textbf{false}\ t\ f = f$$

Let us try to implement a couple of useful operations:

$$\textbf{not}\ x = ?$$
$$\textbf{and}\ x\ y = ?$$
$$\textbf{or}\ x\ y = ?$$

Encoding Booleans

We have:

$$\textbf{true}\ t\ f = t$$
$$\textbf{false}\ t\ f = f$$

Let us try to implement a couple of useful operations:

$$\textbf{not}\ x = x\ \textbf{false}\ \textbf{true}$$
$$\textbf{and}\ x\ y = x\ y\ \textbf{false}$$
$$\textbf{or}\ x\ y = x\ \textbf{true}\ y$$

Encoding Sums

A natural generalization of booleans is the sum type. Recall that $A + B$ can hold a value of type A or a value of type B. Booleans can be seen as a special case of this type where we set $A = B =$ unit. In Haskell, you know this type as `Either`.

As before, we can treat the sum type as an abstract data type with two constructors: **left** and **right**. A value of this type is used by performing case analysis:

$$\text{cases}\ (\textbf{left}\ v)\ l\ r = l\ v$$
$$\text{cases}\ (\textbf{right}\ v)\ l\ r = r\ v$$

Encoding Sums

As with booleans, we can simply pick cases = id and figure out what the encoding should be from there.

$$\textbf{left}\ v = \lambda lr.l\ v$$
$$\textbf{right}\ v = \lambda lr.r\ v$$

But notice that we are not done. For booleans, false and true were already the encoded values. In this case, **left** and **right** are functions that take the inner value (v) and produce the encoding. Moving v to the right gets us:

$$\textbf{left} = \lambda vlr.l\ v$$
$$\textbf{right} = \lambda vlr.r\ v$$

Encoding Sums

$$\textbf{left} = \lambda vlr.l\ v$$
$$\textbf{right} = \lambda vlr.r\ v$$

Let us try to implement a couple of operations:

$$\text{isLeft}\ s = ?$$
$$\text{mapRight}\ f\ s = ?$$

Encoding Sums

$$\textbf{left} = \lambda vlr.l\ v$$
$$\textbf{right} = \lambda vlr.r\ v$$

Let us try to implement a couple of operations:

$$\text{isLeft}\ s = s\ (\lambda x.\textbf{true})\ (\lambda x.\textbf{false})$$
$$\text{mapRight}\ f\ s = s\ (\lambda x.\textbf{left}\ x)\ (\lambda x.\textbf{right}\ (f\ x))$$

As a final note, this encoding can be easily generalized to sums with more than two cases:

$$\text{inj}_i\ v = \lambda x_1 x_2 \ldots x_n.x_i\ v$$

Encoding Products

Let us take a look at a different type: the product. Products bundle multiple values together. You have seen them in the last lecture as pairs $A \times B$, containing both a value of type A and a value of type B.

As an abstract data type, a pair has a single constructor **pair**. A pair can be used in two ways: we can either extract its first element or its second element:

$$\text{fst}\ (\textbf{pair}\ x\ y) = x$$
$$\text{snd}\ (\textbf{pair}\ x\ y) = y$$

Because we have two *different* eliminations, we cannot just set them to *id* as we did before.

Encoding Products

$$\text{fst}\ (\textbf{pair}\ x\ y) = x$$
$$\text{snd}\ (\textbf{pair}\ x\ y) = y$$

Notice that these equations look very similar to booleans: we pick one of two options. Our pair should thus take a function that extracts one of the two elements and apply it to the actual values inside the pair:

$$\textbf{pair}\ x\ y = \lambda e.e\ x\ y$$

We then get:

$$\textbf{pair} = \lambda xye.\ e\ x\ y$$
$$\text{fst}\ p = p\ \textbf{true}$$
$$\text{snd}\ p = p\ \textbf{false}$$

Encoding Products

Let us check that the reduction works as expected:

$$\begin{aligned}
\text{fst}\ (\textbf{pair}\ a\ b) &\to^* \text{fst}\ (\lambda e.e\ a\ b) \\
&\to^* (\lambda e.e\ a\ b)\ (\lambda xy.x) \\
&\to^* (\lambda xy.x)\ a\ b \\
&\to^* a
\end{aligned}$$

If you are not sure how we arrived at the encoding, consider this (somewhat atypical) way to use pairs:

$$\text{elim}\ (\textbf{pair}\ x\ y)\ f = f\ x\ y$$

Set elim $=$ id and you get the encoding for free. Pairs can be defined as both a positive type and a negative type. We might talk about type polarity later. The fst/snd definition is the negative one.

Encoding Products

As before, this approach can be easily generalized and even combined with sums.

$$\textbf{triple} = \lambda abc.\lambda e.e\ a\ b\ c$$

$$\begin{aligned}\textbf{justLeft} &= \lambda v.\lambda lbr.l\ v\\ \textbf{both} &= \lambda v_1 v_2.\lambda lbr.b\ v_1\ v_2\\ \textbf{justRight} &= \lambda v.\lambda lbr.r\ v\end{aligned}$$

Encoding Numbers

In theory, we could encode an `int` as a bounded sum with 2^{32} (2^{64}) different cases, but that is impossible to work with in practice. Since we already know how to encode sums, we can start from the inductive definition of natural numbers and work from there:

```
data Nat = Zero | Succ Nat
```

Applying the same trick as before, we get:

$$\textbf{zero} = \lambda zs.z$$

$$\textbf{succ}\ n = \lambda zs.s\ n$$

Scott Encoding

$$\textbf{zero} = \lambda zs.z$$
$$\textbf{succ}\ n = \lambda zs.s\ n$$

This encoding is known as *Scott encoding*. It is a bit unwieldy, but it works:

$$\textbf{two} = \lambda zs.s\ (\lambda zs.s\ (\lambda zs.z))$$
$$\textbf{isZero}\ n = n\ \textbf{true}\ (\lambda x.\textbf{false})$$

Scott Encoding

Arithmetic operations require the fixed-point combinator.

```
add  Zero     n = n
add (Succ p) n = Succ (add p n)

mul  Zero     n = Zero
mul (Succ p) n = add n (mul p n)
```

$$\textbf{add} = Y\ (\lambda rmn.m\ n\ (\lambda p.\textbf{succ}\ (r\ p\ n)))$$
$$\textbf{mul} = Y\ (\lambda rmn.m\ \textbf{zero}\ (\lambda p.\textbf{add}\ n\ (r\ p\ n)))$$

Try implementing equality!

Church Encoding

Relying on a fixed-point combinator is fine for untyped lambda calculus, but it will be a problem later on. What else can we do?

We can think of a number n as representing the act of doing something n times. In our case, the *something* will be a function.

$$\begin{aligned} \textbf{zero} &= \lambda fx.x \\ \textbf{one} &= \lambda fx.f\ x \\ \textbf{two} &= \lambda fx.f\ (f\ x) \end{aligned}$$

This is known as the Church encoding.

Church Encoding

Church encoding has some advantages. Arithmetic becomes very easy:

$$\textbf{add}\ m\ n = \lambda fx.m\ f\ (n\ f\ x)$$
$$\textbf{mul}\ m\ n = \lambda fx.m\ (n\ f)\ x$$
$$\textbf{exp}\ m\ n = \lambda fx.n\ m\ f\ x$$

(The last one is a bit of a mind-twister; see if you can figure out how it works)

Some operations become considerably harder: predecessor, subtraction, equality.

Encoding Lists

Natural numbers can be generalized to lists; zero becomes the empty list, successor becomes the *cons* cell. As an example, here is the list `[1,2,3]` in both encodings:

$$\textbf{church} = \lambda nc.c\ 1\ (c\ 2\ (c\ 3\ n))$$

$$\textbf{scott} = \lambda nc.c\ 1\ (\lambda nc.c\ 2\ (\lambda nc.c\ 3\ (\lambda nc.n)))$$

Simulating a Turing Machine

Recall that a Turing machine is defined as a tuple $(Q, \Gamma, b, \delta, q_0, F)$ where

- Q is a (finite, non-empty) set of states
- Γ is a (finite, non-empty) set of symbols
- $b \in \Gamma$ is the blank symbol
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \mathbf{2}$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of end states

(Definitions differ, but that usually does not matter too much)

Simulating a Turing Machine

We will simplify the simulation a bit by restricting the symbols to $\Gamma = \{0, 1\}$, with $b = 0$.

A Turing machine operates on a tape that is infinite in both directions. At the start, a finite portion of the tape contains the input; the rest is filled with the blank symbol. The machine begins at position zero and operates as follows:

- Read the symbol s at the current position
- Compute $(q', s', l) = \delta(q, s)$ where q is the current state
- Write s' at the current position
- Move the tape left if $l = 1$, right otherwise
- Set current state to q'
- Stop if $q' \in F$, repeat otherwise

Simulating a Turing Machine

Discussion: How to represent this in lambda calculus? We need states, symbols, tape, end states, and a transition function. What if we want to do it at least somewhat efficiently?