Chapter 3

Recursion in the λ-calculus

We have seen how to implement basic data types and simple operations on them. We want to have a complete programming environment where we can define any computable function. We have already seen, in the definition of plus, mult and exp, how to define iterative functions on Church numerals. Now we see how to implement general recursion.

Church Numerals are Iterators

The idea behind Church numerals is that a number n represent the application of a given function n times. So when we want to iterate an operation a fixed number of times, we just apply the corresponding number to it. This means that functions defined on natural numbers by recursion on their inductive structure can be encoded directly by applying the number to the one-step unfolding of the function. We have seen this already in the definition of the basic arithmetic operations plus, mult, exp.

However, iteration is not as flexible as recursion. In recursion, at each step we are allowed to use both the argument value and the recursive call on its predecessor, while in iteration we have access only to the latter. Take these two definitions of recursive functions, the first computes the exponential on base three 3^n, the second computes the factorial $n!$:

$$\begin{array}{ll} \mathsf{pow}_3 : \mathbb{N} \to \mathbb{N} & \mathsf{fact} : \mathbb{N} \to \mathbb{N} \\ \mathsf{pow}_3\, 0 = 1 & \mathsf{fact}\, 0 = 1 \\ \mathsf{pow}_3\, (n+1) = 3 \cdot (\mathsf{pow}_3\, n) & \mathsf{fact}\, (n+1) = n \cdot (\mathsf{fact}\, n) \end{array}$$

The definition is very similar, but while pow_3 can be directly encoded using iterators, fact is more problematic:

$$\mathsf{pow}_3 = \lambda n.n\, (\mathsf{mult}\, \overline{3})\, \overline{1} \qquad \mathsf{fact} = \lambda n.n\, (\mathsf{mult}\, ?)\, \overline{1}$$

Using n as an iterator means applying the same function n times. This works well with pow_3: we apply n times the function $(\mathsf{mult}\, \overline{3})$, starting with the initial

value $\overline{1}$. But with fact we need to apply a different function ($\mathsf{mult}\,\overline{n}$) at every step. It cannot be done with simple iteration.

One solution is to *remember* the value of the argument by returning it as output together with the output of the function. So we define an auxiliary function $\mathsf{fact}_{\mathsf{aux}}$ such that $\mathsf{fact}_{\mathsf{aux}}\,n = \langle n, \mathsf{fact}\,n\rangle$. We can then extract the value n that we need from it:

$$\mathsf{fact}_{\mathsf{aux}} = \lambda n.n\,(\lambda p.\langle(\mathsf{succ}\,(\mathsf{fst}\,p)), \mathsf{mult}\,(\mathsf{succ}\,(\mathsf{fst}\,p))\,(\mathsf{snd}\,p)\rangle)\,\langle\overline{0}, \overline{1}\rangle.$$

This λ-term is equivalent to the following recursive definition:

$$\begin{aligned}
&\mathsf{fact}_{\mathsf{aux}} : \mathbb{N} \to \mathbb{N} \times \mathbb{N}\\
&\mathsf{fact}_{\mathsf{aux}}\,0 = \langle 0, 1\rangle\\
&\mathsf{fact}_{\mathsf{aux}}\,(n+1) = \langle m+1, (m+1)\cdot r\rangle \quad \text{where } \langle m, r\rangle = \mathsf{fact}_{\mathsf{aux}}\,n.
\end{aligned}$$

To compute $\mathsf{fact}_{\mathsf{aux}}\,(n+1)$, we call it recursively on the predecessor n. The result of this call is a pair $\langle m, r\rangle$. We return $m+1$ and the product of $m+1$ and r. It is easy to see that m will always be equal to n (and r is the factorial of it). But to use iteration, we have to reconstruct n as a returned result, since we cannot use the argument. Finally, the factorial function just returns the second component of the auxiliary function:

$$\mathsf{fact} := \lambda n.\mathsf{snd}\,(\mathsf{fact}_{\mathsf{aux}}\,n).$$

Another example where we need to extend iteration to allow more flexible recursion is the computation of the Fibonacci numbers. Recursively, they are defined as follows:

$$\begin{aligned}
&\mathsf{fib} : \mathbb{N} \to \mathbb{N}\\
&\mathsf{fib}\,0 = 0\\
&\mathsf{fib}\,1 = 1\\
&\mathsf{fib}\,(n+2) = (\mathsf{fib}\,n) + (\mathsf{fib}\,(n+1)).
\end{aligned}$$

In this case, we cannot directly apply iteration because we call the function recursively not just on the predecessor of the argument, but also on the predecessor of the predecessor. The trick to realize this with iteration is, once again, to extend the output to a pair of values, containing respectively the Fibonacci number of the present argument and of its successor:

$$\begin{aligned}
&\mathsf{fib}_{\mathsf{aux}} : \mathbb{N} \to \mathbb{N} \times \mathbb{N}\\
&\mathsf{fib}_{\mathsf{aux}}\,0 = \langle 0, 1\rangle\\
&\mathsf{fib}\,(n+1) = \langle y, x+y\rangle \quad \text{where } \langle x, y\rangle = \mathsf{fib}_{\mathsf{aux}}\,n.
\end{aligned}$$

As λ-term, this function is just an iterator:

$$\begin{aligned}
&\mathsf{fib}_{\mathsf{aux}} = \lambda n.n\,(\lambda p.\langle\mathsf{snd}\,p, \mathsf{plus}\,(\mathsf{fst}\,p)\,(\mathsf{snd}\,p)\rangle)\,\langle\overline{0}, \overline{1}\rangle,\\
&\mathsf{fib} = \lambda n.\mathsf{fst}\,(\mathsf{fib}_{\mathsf{aux}}\,n).
\end{aligned}$$

This technique also gives us a clue to implement the predecessor and subtraction functions on Church numerals, that seemed quite difficult before. The

idea is similar to the Fibonacci one: return two values, the first is the predecessor of the present argument, the second is the predecessor of its successor. Of course that second value is going to be the argument itself, but as before, we need to reconstruct it in the output to have it when we need it.

$$\begin{array}{l}\mathsf{pred}_{\mathsf{aux}} = \lambda n.n\,(\lambda p.\langle \mathsf{snd}\,p, \mathsf{succ}\,(\mathsf{snd}\,p)\rangle)\,\langle \overline{0}, \overline{0}\rangle \\ \mathsf{pred} = \lambda n.\mathsf{fst}\,(\mathsf{pred}_{\mathsf{aux}}\,n).\end{array}$$

Subtraction can be realized by simply iterating the predecessor:

$$\mathsf{minus} = \lambda n.\lambda m.m\,\mathsf{pred}\,n.$$

Other Numeral Systems: The definition of predecessor is correct but inefficient. To compute the predecessor of $(\mathsf{succ}\,\overline{n})$ we must iterate the operation on pairs n times. It would be more efficient if we could just strip the succ away in one step. This is not possible with the Church encoding. But there are other possible ways to represent natural numbers in the λ-calculus. One is the following:

$$\ulcorner 0 \urcorner = \lambda x.x, \qquad \ulcorner n+1 \urcorner = \langle \mathsf{false}, \ulcorner n \urcorner \rangle.$$

As an exercise, try to define the basic arithmetic operations on this encoding. See how much easier it is to define the predecessor.

Another option is to modify Church numerals, enriching them with an extra parameter, so making them recursors rather than just iterators:

$$\widetilde{0} = \lambda f.\lambda x.x, \qquad \widetilde{n+1} = \lambda f.\lambda x.f\,\widetilde{n}\,(\widetilde{n}\,f\,x).$$

TO DO [Show how recursion (for example the factorial and Fibonacci functions, can be done more easily using these alternative number systems.]

TO DO [There representations were all *unary*. Show how to implement a binary representation of natural numbers. Then extend it to numerals in any base.]

General Recursion

So, using clever techniques (auxiliary functions with an extended output) we can realize full recursion. However, not all functions on natural numbers are recursive on the size of the number: the recursive calls may be applied to values that are not the direct predecessor of the input and may even be bigger. Some general recursive functions are not even total: they may diverge for some input values.

As an example, consider the following variant of the *hailstone sequence*:

$$\mathsf{hail}\,n = \begin{cases} 0 & \text{if } n \leq 1 \\ \mathsf{hail}\,(n/2)+1 & \text{if } n > 1, n \text{ even} \\ \mathsf{hail}\,(3n+1)+1 & \text{if } n > 1, n \text{ odd} \end{cases}$$

Traditionally, starting from a number n, the hailstone sequence goes through the numbers obtained by the following process: if n is 0 or 1 stop; if n is larger than 1 and odd, the next number in the sequence is the half of n; if n is larger than 1 and odd, the next number is three times n plus 1. The function hail returns the length of the hailstone sequence.

Notice that in the last case the recursive call is $\mathsf{hail}\,(3n+1)$, so we must calculate the function on an argument that is larger that the original input. This cannot be realized with normal recursion. We call it *general recursion*, meaning that recursive calls are unrestricted. This of course does not guarantee that the computation will never end. In the specific case of the hailstone sequence, it has been verified that it terminates for all initial values smaller than 2^{60}. But nobody knows whether it always terminates for any value. This is still a mathematical mystery, known as the *Collatz Conjecture*.

How can we implement hail in the λ-calculus? Surely we can't hope to do it with simple recursion: that would automatically imply that the function always terminates.

If we want to use the λ-calculus as a complete programming language, there should be a way to define any computable function, even non-total ones. This is indeed possible. In fact there is a single λ-term, called *the* Y *combinator*, that allows us to use unrestricted recursion:

$$\mathsf{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)).$$

This definition is inspired by the non-normalizing term ω. In fact we have that $\mathsf{Y}\,\mathsf{id} \rightsquigarrow^* \omega$. The most striking property of Y is that it computes a *fixed point* for every term F:

$$\mathsf{Y}\,F \rightsquigarrow^* F\,(\mathsf{Y}\,F).$$

This is not exactly true: to be precise, $\mathsf{Y}\,F \rightsquigarrow (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) =: \mathsf{fix}_F$ and $\mathsf{fix}_F \rightsquigarrow^* F\,(\mathsf{fix}_F)$. If we keep reducing, we get an infinite sequence of applications of F:

$$\mathsf{fix}_F \rightsquigarrow^* F\,\mathsf{fix}_F \rightsquigarrow^* F\,(F\,\mathsf{fix}_F) \rightsquigarrow^* F\,(F\,(F\,\mathsf{fix}_F)) \rightsquigarrow^* \cdots$$

To define a recursive function, we just need to encode a single step of it as a term F and then use Y to iterate that step as many times as it is necessary to get a result. Here is, for example, the definition of the factorial:

$$\begin{aligned}&\mathsf{fact}_{\mathsf{step}} := \lambda f.\lambda n.\mathsf{if}\,(\mathsf{isZero}\,n)\,\overline{1}\,(\mathsf{mult}\,n\,(f\,(\mathsf{pred}\,n)))\\ &\mathsf{fact} := \mathsf{Y}\,\mathsf{fact}_{\mathsf{step}}.\end{aligned}$$

We can also define the function that measures the length of the hailstone sequences. (Exercise: implement the test functions leq that checks the less-or-equal relation, even that checks divisibility by two, and the division by two half.)

$$\begin{aligned}\mathsf{hail} = \mathsf{Y}\,\lambda h.\lambda n.\mathsf{if}\ \ &(\mathsf{leq}\,n\,\overline{1})\\ &\overline{0}\\ &\mathsf{if}\ \ (\mathsf{even}\,n)\\ &\quad(h\,(\mathsf{half}\,n))\\ &\quad(h\,(\mathsf{succ}\,(\mathsf{mult}\,\overline{3}\,n)))\end{aligned}$$

The combinator Y is one of the most important constructions in the λ-calculus. Using it, we can show that all computable functions can be represented. Therefore the λ-calculus is a Turing-complete language: every function that can be computed in any known model of computation (Turing machines, μ-recursive functions, the von Neumann architecture, all known programming languages) can be realized by a λ-term.

The statement that this notion of computation is universal, that is, nobody will ever find a way to compute something that cannot be realized in one and all of these models, is know as *the Church-Turing Thesis*.

Infinite sequences

We have seen how tuples and lists can be represented by λ-terms. One of the applications of the Y combinator is that we can also represent infinite sequences, know in computer science as *streams*.

The notation for streams uses a *cons* constructor $\lhd$. So a generic streams is denoted by

$$a_0 \lhd a_1 \lhd a_2 \lhd a_3 \lhd \cdots.$$

Intuitively, a stream is implemented as iterated pairing:

$$\langle a_0, \langle a_1, \langle a_2, \langle a_3, \cdots \rangle \rangle \rangle \rangle.$$

However, obviously we cannot write out this whole term explicitly. The best we can do is to define a λ-term t that reduces to partial unfoldings of the stream:

$$\begin{aligned} t &\rightsquigarrow^* \langle a_0, t_0 \rangle \\ &\rightsquigarrow^* \langle a_0, \langle a_1, t_1 \rangle \rangle \\ &\rightsquigarrow^* \langle a_0, \langle a_1, \langle a_2, t_2 \rangle \rangle \rangle \\ &\rightsquigarrow^* \langle a_0, \langle a_1, \langle a_2, \langle a_3, t_3 \rangle \rangle \rangle \rangle \\ &\rightsquigarrow^* \cdots. \end{aligned}$$

This term cannot be normalizing, but has to reduce forever, generating all the elements of the stream in the process. We can construct it by a use of the fixed point combinator.

For example, let's see how to realize the stream of all natural numbers, $\mathsf{nats} = 0 \lhd 1 \lhd 2 \lhd 3 \lhd \cdots$. More generally, we want to define a function $\mathsf{natsFrom}$ that generates the sequence of naturals starting from a given input: $\mathsf{natsFrom}\, n = n \lhd (n+1) \lhd (n+2) \lhd (n+3) \lhd \cdots$. This can be easily realized using the Y combinator:

$$\mathsf{natsFrom} = \mathsf{Y}\,(\lambda f.\lambda n.\langle n, f\,(\mathsf{succ}\, n)\rangle), \qquad \mathsf{nats} = \mathsf{natsFrom}\,\overline{0}.$$

The *head* function, returning the first element of a stream, and the *tail* function, returning the rest of the stream, are just the first and second projections of pairs:

$$\mathsf{head} = \mathsf{fst}, \qquad \mathsf{tail} = \mathsf{snd}.$$

We can easily define point-wise operations on streams. For example, adding two streams of natural numbers one element at a time can be specified by a recursive equation:

$$\sigma \oplus \tau = (\mathsf{head}\,\sigma + \mathsf{head}\,\tau) \lhd (\mathsf{tail}\,\sigma \oplus \mathsf{tail}\,\tau)$$

which we realize by an application of the fixed point combinator:

$$\mathsf{sum}_{\mathsf{stream}} = \mathsf{Y}\,\lambda f.\lambda x.\lambda y.\langle \mathsf{plus}\,(\mathsf{head}\,x)\,(\mathsf{head}\,y), f\,(\mathsf{tail}\,x)\,(\mathsf{tail}\,y)\rangle$$

This gives us a clever way to define the streams of Fibonacci numbers. Intuitively, we can specify it by the following recursive equation:

$$\mathsf{fibs} = 0 \lhd 1 \lhd (\mathsf{fibs} \oplus (\mathsf{tail}\,\mathsf{fibs}))$$

which can again be realized by an application of the fixed point combinator:

$$\mathsf{fibs} = \mathsf{Y}\,\lambda s.\langle \overline{0}, \langle \overline{1}, \mathsf{sum}_{\mathsf{stream}}\,s\,(\mathsf{tail}\,s)\rangle\rangle.$$

Evaluation Strategies

As we have seen, a λ-term may contain several redexes. We can choose which one to reduce first. When we perform a step of reduction, new redexes may appear, old redexes may be duplicated or disappear. However, the confluence theorem guarantees that two different reduction paths can always be continued to a common reduct. A corollary is *uniqueness of normal forms*: if a term is reduced to normal form using two different reduction paths, the normal form at the end of the two paths is the same.

However, nothing guarantees that we will reach a normal form using any path, even when a normal form exists. A simple illustration is the following term:

$$(\lambda x.\lambda y.x)\,\mathsf{id}\,\omega.$$

This term contains two redexes. The first one is the top one involving the variable x. If we reduce it and then reduce the new redex involving the variable y, we obtain a normal form:

$$(\lambda x.\lambda y.x)\,\mathsf{id}\,\omega \rightsquigarrow (\lambda y.\mathsf{id})\,\omega \rightsquigarrow \mathsf{id}.$$

However, if we try to reduce the redex inside ω, we obtain the same term, and we could stubbornly try to reduce that same redex again in an infinite reduction path:

$$(\lambda x.\lambda y.x)\,\mathsf{id}\,\omega \rightsquigarrow (\lambda x.\lambda y.x)\,\mathsf{id}\,\omega \rightsquigarrow (\lambda x.\lambda y.x)\,\mathsf{id}\,\omega \rightsquigarrow \cdots.$$

So whether we eventually reach a normal form or not may depend on our choice of redex.

There are several main *reduction strategies*, ways of choosing which redex to reduce first. Whether we reach a normal form and the complexity of the computation (how many reduction steps we need to do) depends on the strategy.

Full β-reduction: Reduce any redex you want. If we reduce all the redexes present in the original term before reducing the new ones created by reduction steps, we are guaranteed to find the normal form, if it exists. However, we have to take care of the fact that the original redexes may be duplicated into many copies and we must reduce each of them. If we use a term-graph representation this is not a problem: with sharing it's not the subterms containing the redexes that are duplicated, but just the pointer to them. An extended notion of reduction may allow us to reduce all redexes present in a term in just one step.

Normal Order: always reduce the leftmost redex (the top redex in the abstract syntax tree representation). This is a *normalizing strategy*: if a normal form exists, this strategy will find it.

Call-by-Name: Never reduce under an abstraction, so a term of the form $\lambda x.t$ does not reduce, no matter how many redexes t contains; we consider it already a value. On the other hand, when we have an application $(f\,u)$, we first reduce f until it becomes an abstraction (if ever), $f \rightsquigarrow^* \lambda x.t$; then we reduce the top redex, $(\lambda x.t)\,u \rightsquigarrow t[x := u]$; then we continue reducing the resulting term. The reason this strategy is designated as *call-by-name* is that, when applying a function, we make the substitution $[x := u]$ immediately, before reducing u to a value, so we use the *name* u, that is, the syntactic form, rather than its value.

Call-by-Value: As in the previous one, but when reducing the application $(\lambda x.t)\,u$ we must first reduce u to a value and only afterwards reduce the main redex. As we mentioned in the previous point, we consider every λ-abstraction a value. In richer systems, where we have actual native data types, there will be also more precise notions of what a value is. For example, a value of a number type will be a numeral.

In functional programming languages there is a divide between *strict* evaluation, which is a variant of the call-by-value reduction strategy, and *lazy* evaluation, which is a variant of the call-by-name strategy. OCaml is a popular strict programming language. Haskell is a popular lazy programming language.