

Department of Computer Science  
Australian National University

COMP3610  
Principles of Programming Languages

## Fixpoint Theory of Recursive Function Definitions

Clem Baker-Finch  
July 31, 2007

## Contents

1	Introduction	1
2	Computational approach	2
3	Complete partial orders	4
4	Monotonic functions	6
5	Computational approach (continued)	7
6	Fixed points	8
7	Computation rules	9
7.1	Reduction strategies in the $\lambda$ -calculus	10
7.2	Computation rules for recursive function definitions	11

This section of the *Principles of Programming Languages* course gives an introduction to the fixpoint theory of recursive function definitions. The aim is to give a precise way of specifying the mathematical function that is “defined” by a recursive definition in a programming language such as Haskell. Of course, it is most important that the function so defined corresponds to our intuitive computational interpretation of the program. Note that these notes assume some familiarity with the notions of *bottom* for non-termination and *strictness* in the context of Haskell.

## 1 Introduction

Most of us are comfortable with using recursive equations to define a function — we think of it as a *rule* for computing a result from given arguments. But if we think of it merely as an *equation*, it does not necessarily uniquely define a function. Consider the following equation:

$$f x = \text{if } x == 0 \text{ then } 1 \text{ else } f(x+1) \quad (1.1)$$

How does this *define*  $f$ ? That is, what is the *graph* of the function (as a set of ordered pairs)? Before attempting to answer that question, let's look at something simpler — the equation  $z + 4 = 6$ . We can 'solve' this equation to deduce  $z = 2$ . What this really means is that if we *substitute* 2 for  $z$ , the equation is *true*.

Going back to the equation for  $f$  above, in a mathematical sense, all we can do is 'solve' for  $f$ . In other words, find a function that can be substituted for  $f$  which makes the equation true. In fact, there is an infinite number of such functions, such as the following  $f_n$  for all  $n$ :

$$f_n x = \begin{cases} 1 & \text{if } x \leq 0 \\ n & \text{if } x > 0 \end{cases}$$

Alternatively we could use lambda notation to write them as anonymous functions:

$$\lambda x. \text{if } x \leq 0 \text{ then } 1 \text{ else } n$$

But if you think about *computing* with a function defined as in equation (1.1) you should expect to have the function:

$$\lambda x. \text{if } x \leq 0 \text{ then } 1 \text{ else } \perp$$

Recall that  $\perp$  represents non-termination as a special value. Try it in Haskell. Note that, since the possibility of non-termination is pervasive in computing, we are routinely dealing with *partial* functions. It is convenient to use a symbol such as  $\perp$  to signify undefinedness as a *value*. More will be said about this in Section 3.

As another example, consider the equation:

$$f x = \text{if } f x == 0 \text{ then } 1 \text{ else } 0$$

Your first reaction may be that this is nonsense and there is no such  $f$ , but in fact the totally undefined function  $\lambda x. \perp$  satisfies the equation.

One more example:

$$f x = f x$$

These examples may look silly, but hopefully they illustrate the point. In this case it is obvious that *all* functions satisfy the equation, but computational intuition again suggests that the totally undefined function is 'the' solution for  $f$ .

What we need is some way to specify *which* solution of the recursive equation is *intended*. In other words, which one corresponds to our computational intuition? That is what these notes attempt to explain.

### Exercise 1.1

How many solutions do each of the following equations have? Which correspond to your computational intuition?

$$\begin{aligned} f x &= \text{if } f x == 0 \text{ then } 0 \text{ else } 1 \\ f(x, y) &= \text{if } x == y \text{ then } y + 1 \text{ else } f(x, f(x-1, y+1)) \end{aligned}$$

## 2 Computational approach

We will begin by attempting to clarify our computational intuitions. Quite a bit of machinery will be required to make all this work in the mathematical sense, but let's start informally. To simplify matters, we will usually work in the domain of *natural numbers*  $\mathbb{N}$  (unless otherwise noted). All the results carry over to other domains.

When we (as programmers or computer scientists) write recursive function definitions, we usually have a *computational interpretation* in mind, based on the idea of rewriting instances of the left hand side to the right hand side. Indeed, some functional programming languages use an arrow like  $\Rightarrow$  rather than  $=$  to emphasise this idea of *unfolding*. For example, to *compute* the value of  $f3$  using the definition in (1.1) above, we proceed as follows:

$$\begin{aligned} f 3 &\Rightarrow \text{if } 3 == 0 \text{ then } 1 \text{ else } f 4 \\ &= f 4 \\ &\Rightarrow \text{if } 4 == 0 \text{ then } 1 \text{ else } f 5 \\ &= f 5 \\ &\Rightarrow \dots \end{aligned}$$

We can do the unwinding symbolically:

$$\begin{aligned} f x &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else } f(x+1) \\ &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else if } x+1 == 0 \text{ then } 1 \text{ else } f(x+2) \\ &= \text{if } x == 0 \text{ then } 1 \text{ else } f(x+2) \\ &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else } f(x+3) \\ &\Rightarrow \dots \end{aligned}$$

The factorial function is a familiar example.

$$fac x = \text{if } x == 0 \text{ then } 1 \text{ else } x \times fac(x-1) \quad (2.1)$$

In this case, it so happens that there is only one function that satisfies this equation for  $fac$ , but how do I know that?

Our computational intuition is the following unfolding:

$$\begin{aligned} fac x &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else } x \times fac(x-1) \\ &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else } x \times (\text{if } (x-1) == 0 \text{ then } 1 \text{ else } (x-1) \times fac((x-1)-1)) \\ &= \text{if } x == 0 \text{ then } 1 \text{ else if } x == 1 \text{ then } 1 \text{ else } x \times (x-1) \times fac(x-2) \\ &\Rightarrow \text{if } x == 0 \text{ then } 1 \text{ else if } x == 1 \text{ then } 1 \\ &\quad \text{else if } x == 2 \text{ then } 2 \text{ else } x \times (x-1) \times (x-2) \times fac(x-3) \\ &\Rightarrow \dots \end{aligned}$$

Notice that each unfolding is a better and better approximation to the intended function. If we want to discover the (graph of the) function that corresponds to our computational intuition, we need to *formalise this unfolding in a mathematical sense, rather than an operational sense*.

The first key insight is to abstract out a *higher-order function that represents the right hand side of the definition*. To make thing more concrete I'll write it in Haskell, so you don't think there's any magic happening. Firstly, the basic definition of the factorial function:

```

fac :: Int -> Int
fac x = if x==0
      then 1
      else x * fac (x-1)

```

Here is an alternative definition, using a higher-order function `facF`. Notice that `facF` is not recursive. You should also notice that `facF` really has nothing to do with `fac` — it's just a function. For example, we can evaluate the expression `facF (*100) 4` to give 1200.

```

facF :: (Int-> Int) -> (Int -> Int)
facF = \g x ->
      if x==0
      then 1
      else x * g (x-1)

fac' :: Int-> Int
fac' = facF fac'

```

Now the unfolding of the right hand side we talked about above is just repeated application of `facF`.

```

fac => facF fac
    => facF (facF fac)
    => facF (facF (facF fac))
    => ...

```

The only remaining question is where do we *start* the unfolding? Clearly, if we don't do any unfolding of `fac` we don't get any answers so it seems reasonable to start with `⊥`. This is the second key insight.

Remember that we can represent `⊥` in Haskell with the definition `bot = bot`. This is a bit informal, but the first few unfoldings are:

```

facF bot           => \x -> if x==0 then 1 else bot
facF (facF bot)   => \x -> if x==0 then 1
                  else if x==1 then 1 else bot
facF (facF (facF bot)) => \x -> if x==0 then 1
                  else if x==1 then 1
                  else if x==2 then 2 else bot

```

Notice again that each time we unfold (i.e. apply `facF`) we get a better *approximation* to `fac`. Of course, we can code up this chain of approximations in Haskell:

```

chain :: (a -> a) -> [a]
chain fF = iterate fF bot

```

and evaluate expressions like `(chain facF !! 4) 3` (the 4th approximation to `fac 3`) and `(chain facF !! 3) 6` (the 3rd approximation to `fac 6`).

**Here we have the main idea:** *it seems reasonable to deduce that the function that corresponds to our computational intuition is the **limit** of this sequence of approximations.*

In Haskell, this limit is equal to the last element of the list generated by `chain`, so it might be tempting to write:

```

lfp :: (a -> a) -> a
lfp fF = last (chain fF)

```

but of course, it's an infinite list, so we will never get an answer.

Mathematically, we would like to define:

$$fac = \lim_{n \rightarrow \infty} facF^n \perp$$

(Note that  $n$  repeated applications of a function  $f$  is the same as an  $n$ -fold composition of  $f$ .)

But what do we *mean* by the limit? The notion of a limit only makes sense in ordered domains. For our purposes, the appropriate ordering is in terms of “definedness”, and the idea of a limit is defined in terms of the least upper bound. The next section aims to give some more insights into these ideas.

### Exercise 2.1

Repeat this process for the function  $f$  defined at (1.1) above: write it in terms of a higher order function  $F$  such that  $f = Ff$ . Calculate the “limit” of the chain  $\perp, F\perp, F^2\perp, \dots$ . You might like to code it in Haskell.

## 3 Complete partial orders

To allow us to talk about ‘undefined’ as a value, we have assumed a special element `⊥` in every domain of values. You may have been wondering why this symbol is called *bottom*. The following definition should make it clear.

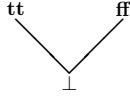
### Definition 3.1

In fact, this addition of `⊥` is called ‘lifting’ and comes with a *partial ordering* relation: *less defined than or equal* which is denoted  $\sqsubseteq$ . A partial ordering is a relation that is reflexive ( $d \sqsubseteq d$ ), transitive ( $(d_0 \sqsubseteq d_1) \wedge (d_1 \sqsubseteq d_2) \Rightarrow (d_0 \sqsubseteq d_2)$ ), and antisymmetric ( $(d_0 \sqsubseteq d_1) \wedge (d_1 \sqsubseteq d_0) \Rightarrow (d_0 = d_1)$ ). With respect to  $\sqsubseteq$ , `⊥` is the least element.

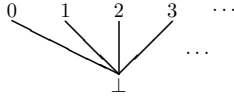
On the simple flat domains such as  $\mathbb{N}$  and  $\mathbb{B}$  this ordering is almost trivial to define:

- $\forall n : \mathbb{N}, n \sqsubseteq n$
- $\forall n : \mathbb{N}, \perp \sqsubseteq n$
- $\perp \sqsubseteq \perp$

The ordering can be presented as Hasse diagrams, where edges indicate that the connected elements are related by  $\sqsubseteq$  and the lower one is the lesser of the two. The following diagram shows the partial ordering on  $\mathbb{B}$ :



Similarly, the Hasse diagram for  $\mathbb{N}$  is as follows:



From now on we will assume that all domains come equipped with a  $\perp$  representing the least defined value and the  $\sqsubseteq$  partial order. The ordering on constructed domains such as (cartesian) products and function spaces is a little less straightforward but is induced by the ordering on the flat domains.

We will leave function spaces to the next section. Given two partial orders  $D_0$  and  $D_1$ , the partial order on their product  $D_0 \times D_1$  is given by  $(x, y) \sqsubseteq (x', y')$  iff  $x \sqsubseteq x'$  and  $y \sqsubseteq y'$ . Note that the least element is therefore  $(\perp, \perp)$ . In fact there are other possible ways to construct the product, but they will not concern us here.

To formalise the ideas outlined in Section 2 regarding the limit of a sequence of approximations, we need the notion of *least upper bound*.

### Definition 3.2

Given a partial ordering  $\sqsubseteq$  on a set  $D$ , for all subsets  $X \subseteq D$ , the *least upper bound (lub)* of  $X$  (if it exists) is an element of  $D$ , denoted  $\bigsqcup X$ , such that:

- i. for all  $x \in X$ ,  $x \sqsubseteq \bigsqcup X$
- ii. for all  $d \in D$ , if for all  $x \in X$ ,  $x \sqsubseteq d$  then  $\bigsqcup X \sqsubseteq d$

### Definition 3.3

Given a partial order  $\sqsubseteq$  on a set  $D$ , a subset  $X$  of  $D$  is a *chain* iff  $X$  is non-empty and for all  $a, b \in X$  either  $a \sqsubseteq b$  or  $b \sqsubseteq a$ .

Chains may be either finite or infinite. Essentially, they are a sequence of elements which contain consistent information. For example in the domain  $\mathbb{N} \times \mathbb{N}$  with the usual ordering,  $(\perp, \perp) \sqsubseteq (3, \perp) \sqsubseteq (3, 5)$  is a chain. On the other hand,  $(3, \perp)$  and  $(7, 1)$  cannot be elements of the same chain because they are not related by  $\sqsubseteq$ . We are yet to define the  $\sqsubseteq$ -ordering on functions but you will see in the next section that an example of an infinite chain is the sequence of approximations to the factorial function discussed in Section 2.

As hinted in definition 3.2 lubs do not always exist, but our stated intention is that a recursively defined function like *fac* is taken to be the lub of  $\perp, \text{fac}F\perp, \text{fac}F^2\perp, \dots$ . To make this valid, we will require all domains to be *pointed complete partial orders*, defined as follows:

### Definition 3.4

A partially ordered set  $D$  is a *complete partial order (cpo)* iff every chain in  $D$  has a lub in  $D$ . If  $D$  is a cpo and also has a least element then it is a *pointed* complete partial order.

It is possible to ensure that all the domains we use are pointed cpos by insisting that they are always constructed in a way that guarantees the structure is maintained. In simple cases this is straightforward, for example, lifted flat domains; the pointwise ordering on product tuples and so on. On the other hand, it is even acceptable to define domains recursively, e.g.  $List = \{\ [], \ (\mathbb{N} \times List)\}$  defines a domain of finite and infinite lists of integers. These recursive data domains arise in Haskell as recursive user-defined data types. Such material is beyond the scope of this introductory course.

Our computational intuition, developed in section 2, was that a recursive function definition of the form:

$$f = Ff$$

should define a function  $f = \lim_{n \rightarrow \infty} F^n \perp$ . The standard definition of a limit is in terms of least upper bounds — see any elementary analysis or calculus textbook.

So, if we have a sequence  $\{F^n \perp \mid n \geq 0\}$  then, *provided it is a chain*, we have:

$$\lim_{n \rightarrow \infty} F^n \perp = \bigsqcup \{F^n \perp \mid n \geq 0\}$$

*The whole point of the construction of cpos in this section is to ensure that lubs exist, and consequently that limits exist.*

All that remains is to ensure that sequences like  $\perp, F\perp, F^2\perp, \dots$  always form a chain, and that is the topic to which we now turn our attention.

## 4 Monotonic functions

In fact, what we require for the functions in a recursive definition to be well-behaved is that they are *monotonic*.

### Definition 4.1

A function  $f$  is *monotonic* if the following condition is satisfied for all  $d_0, d_1$ :

$$d_0 \sqsubseteq d_1 \Rightarrow fd_0 \sqsubseteq fd_1$$

That is, the more ‘information’ there is in the argument, the more ‘information’ there must be in the result. Put another way, a function cannot return a more defined result for a less defined argument. In the simple case of functions on flat domains such as  $\mathbb{N}$  and  $\mathbb{B}$ , monotonicity comes down to the following requirement:

$$\text{either } f\perp = \perp \\ \text{or } f \text{ is constant}$$

In effect, monotonic functions on flat domains are either strict or ignore their argument.

### Exercise 4.2

There are 27 functions from  $\mathbb{B}$  to  $\mathbb{B}$  (since  $\mathbb{B}$  has 3 elements and  $3^3 = 27$ ) but only 11 of them are monotonic. Which ones are they? (Give their graphs, e.g. as sets of ordered pairs.)

The requirement of monotonicity makes sense from a computational viewpoint, too: if we permitted non-monotonic functions then it would be possible to specify *uncomputable* program constructs, for example, admitting tests for termination.

```
-- This won't even pass the type checker, but you get the idea...
halts f x = if f x /= bot then True else False
```

The following definition tells how to extend the ‘less defined than or equal’ partial ordering to function spaces.

### Definition 4.3

Consider two functions  $f, g : D_0 \rightarrow D_1$ . Then  $f \sqsubseteq g$  iff for all  $d : D_0$ ,  $fd \sqsubseteq gd$ .

The totally undefined function  $\lambda d. \perp$  is the least element of this order. Thus we may occasionally just write  $\perp$  for this function, which is a little ambiguous but should be clear from context.

### Exercise 4.4

What is the partial ordering on the 11 monotonic functions identified in exercise 4.2?

Our aim in this section was to show that the sequences of approximations  $\{f^n \perp \mid n \geq 0\}$  always formed a chain. The following result summarises.

### Proposition 4.5

If  $f$  is monotonic then  $\{f^n \perp \mid n \geq 0\}$  is a chain.

**Proof** By induction on  $n$ .

Base case:  $\perp \sqsubseteq f \perp$  by definition.

Inductive case: If  $f^n \perp \sqsubseteq f^{n+1} \perp$  then since  $f$  is monotonic,  $f(f^n \perp) \sqsubseteq f(f^{n+1} \perp)$ . ■

## 5 Computational approach (continued)

The intuitions we developed in section 2 was that the function defined by a recursive equation, which also corresponds to our computational intuition, was the limit of a particular chain of approximations. We are now able to be more precise about this idea, using the ideas developed in section 3 and section 4.

Fortunately, it so happens that all functions constructed by composition from a base set of monotonic functions are also monotonic. Therefore, if our language of recursive definitions fits this requirement, all the results about chains and limits apply without further consideration.

Given a recursive function definition, we first cast it in the form:

$$f = Ff$$

Assuming  $F$  is monotonic, we immediately have:

$$f = \bigsqcup \{F^n \perp \mid n \geq 0\}$$

## 6 Fixed points

In this section we will take a slightly different point of view of recursive definitions, eventually showing that it corresponds exactly with the approach developed in the previous sections.

### Definition 6.1

A *fixpoint* (or a *fixed point* if you prefer) of a function  $f$  is any value  $x$  such that  $fx = x$ .

If you think about recursive definitions, especially when they are written as

```
fac = facF fac
```

it is clear that **fac** is a fixpoint of some higher-order function **facF**. Similarly, the functions that satisfy equation (1.1) are all fixpoints of  $F = \lambda g. \lambda x. \text{if } x == 0 \text{ then } 1 \text{ else } g(x+1)$ . This is easy to see since equation (1.1) is *exactly* the same as  $f = Ff$ .

However, the fundamental problem remains that there may be 0, 1 or many such fixpoints. *Which one corresponds to our computational intuition?*

If you carefully inspect a variety of examples, it appears that the difference between the desired fixpoint and the others is that the others may give a proper result at points where the desired function is undefined. For example, among the solutions to equation (1.1), the one we want is less defined than or equal to all the others:

$$(\lambda x. \text{if } x == 0 \text{ then } 1 \text{ else } \perp) \sqsubseteq (\lambda x. \text{if } x == 0 \text{ then } 1 \text{ else } n)$$

for all  $n$ . In fact, this is the key insight: the fixpoint that corresponds to our computational intuition is always *the least defined fixpoint*.

### Definition 6.2

$f$  is the *least fixpoint* of  $F$  if  $f = Ff$  and for all other fixpoints  $g$  of  $F$  (i.e.,  $g$  where  $g = Fg$ ),  $f$  is less defined than or equal to  $g$  (i.e.,  $f \sqsubseteq g$ ).

The following famous result brings all of the ideas in these notes together:

### Theorem 6.3 (Kleene’s first recursion theorem)

If  $D$  is a pointed cpo and  $f : D \rightarrow D$  is monotonic, then  $f$  has a unique least fixpoint which we denote **fix**  $f$ . Furthermore:

$$\mathbf{fix} f = \bigsqcup \{f^k(\perp) \mid k \geq 0\}$$

Thus we have a precise definition of the solution of recursive function definitions which corresponds to our computational intuitions. Theorem 6.3 gives a precise way of *finding* the least fixpoint. Furthermore, it so happens that **fix** is a monotonic function.

By this result, we may write  $\mathbf{fac} = \mathbf{fix}(\lambda g. \lambda x. \text{if } x == 0 \text{ then } 1 \text{ else } x \times g(x-1))$  as a non-recursive definition of the factorial function. Furthermore, (writing  $\mathbf{facF}$  for  $\lambda g. \lambda x. \text{if } x == 0 \text{ then } 1 \text{ else } x \times g(x-1)$ ) we have:

$$\mathbf{fac} = \bigsqcup \{\mathbf{facF}^k(\perp) \mid k \geq 0\}$$

precisely matching our intuitive expectations.

The function `lfp` we discussed in section 2 was a doomed attempt to define a Haskell implementation of this limiting process. (You may now deduce that `lfp` stands for least fixed point.)

We can code a `fix` function in Haskell, but it isn't particularly satisfying because it merely reconstructs the recursion:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

```
fac'' :: Int -> Int
fac'' = fix facF
```

#### Exercise 6.4

The equation:

$$f(x) = \text{if } x == 0 \text{ then } 1 \text{ else } 2 \times f(f(x - 1))$$

has exactly 3 fixpoints:

```
λx.if x == 0 then 1 else ⊥
λx.case x of
  0 ⇒ 1
  1 ⇒ 0
  2 ⇒ 2
  3 ⇒ 4
  otherwise ⇒ ⊥
λx.case x of
  0 ⇒ 1
  1, 4, 7... ⇒ 0
  2, 5, 8... ⇒ 2
  3, 6, 9... ⇒ 4
```

Convince yourself that these are solutions and that they are the only solutions. Which is the least fixpoint?

## 7 Computation rules

We have now settled on a precise definition of the function that is defined by a recursive equation and argued that it corresponds to our computational intuition. There is one last question to be addressed:

*If we mechanise the computation process, how can we be sure to compute results that correspond to the least fixpoint?*

More precisely, will a given mechanical process (i.e. a computation rule) always terminate for arguments where the least fixpoint solution is defined?

We will put recursion aside for the moment and consider the general question of the order of evaluation for the  $\lambda$ -calculus.

### 7.1 Reduction strategies in the $\lambda$ -calculus

Recall that  $\beta$ -reduction for the lambda calculus is the following rule:

$$(\lambda x.M)N \rightarrow M[x := N]$$

This can be applied to *any redex* that appears in a term, for example:

$$\begin{aligned} (\lambda x.M)((\lambda y.P)N) &\rightarrow M[x := ((\lambda y.P)N)] \\ (\lambda x.M)((\lambda y.P)N) &\rightarrow \lambda x.M(P[y := N]) \end{aligned}$$

A consequence of the *Church-Rosser property*

If  $M = N$  then there is a term  $P$  such that  $M \rightarrow P$  and  $N \rightarrow P$ .

is that no matter which reductions have already been performed on some term  $M$  it is always *possible* to reach the normal form for  $M$  (provided it exists). In general that means there *exists* a reduction sequence that reaches normal form. It doesn't mean that *any* reduction sequence will reach normal form. Recall that  $\Omega = (\lambda x.xx)(\lambda x.xx)$   $\beta$ -reduces to itself and thus has no normal form. Now consider the following example:

$$(\lambda x.\lambda y.y) \Omega \tag{7.1}$$

which has normal form  $\lambda y.y$ . However, so long as we continue to choose to reduce the  $\Omega$  redex, we will not achieve the normal form.

If we wish to mechanise reduction (e.g. write a reduction program) we must decide on a way of choosing *which redex* to reduce next. In light of the example above, this is the key question.

#### Which redex?

Obviously there are many ways to specify which redex to reduce next, including parallel and non-deterministic approaches, which we will not consider here. There are two strategies that are particularly important:

- **applicative order**, where we always choose the leftmost-innermost redex
- **normal order**, where we always choose to reduce the leftmost-outermost redex

An *innermost redex* is a redex that *contains* no other redex.

An *outermost redex* is a redex that *is contained by* no other redex.

Essentially, *applicative order* evaluates the argument of an application first, whereas *normal order* substitutes the unevaluated argument into the body of the redex. Therefore, applicative order has the advantage of evaluating the argument exactly once, whereas normal order may generate several copies of the argument, each to be separately evaluated. On the other hand, the argument may not be needed at all, in which case the applicative order will do some unnecessary work — obviously a disadvantage.

### Normalising strategies

A reduction strategy for the  $\lambda$ -calculus is said to be **normalising** if, for every term that has a normal form, the strategy achieves that normal form.

- The *normal order* strategy (denoted  $\xrightarrow{n}$ ) is normalising
- The *applicative order* strategy (denoted  $\xrightarrow{a}$ ) is not normalising

For example, the term (7.1) normalises in a single step under the normal order strategy:

$$(\lambda x. \lambda y. y) \Omega \xrightarrow{n} \lambda y. y$$

However, under applicative order it will never reach normal form:

$$(\lambda x. \lambda y. y) \Omega \xrightarrow{a} (\lambda x. \lambda y. y) \Omega \xrightarrow{a} (\lambda x. \lambda y. y) \Omega \xrightarrow{a} \dots$$

## 7.2 Computation rules for recursive function definitions

Recursive definitions are a little more complex than  $\lambda$ -terms; they include base functions and operators, conditional expressions, function definitions and explicit recursion, but we will see that the general ideas of reduction strategies and normalisation carry over.

First we assume (require) that all base functions are *strict*. Recall that a function  $f$  is strict iff  $f \perp = \perp$ . That is, if an argument fails to terminate, the application also does not terminate. (In particular, for example,  $0 \times \perp$  is  $\perp$ , not 0.)

We also require that conditional expressions are strict in their first (boolean) subexpression.

### Exercise 7.1

Obviously, conditional expressions cannot be strict in the other two subexpressions (the **then** and **else** parts). Why? What would the function  $fac$  defined in (2.1) be if conditionals were strict?

Now we can assume (require) that other functions — those the programmer has defined by equations — are either *strict* or *non-strict*.

If they are to be **strict**, then in an application like  $f e$  the argument  $e$  is evaluated *before* substitution into the body of  $f$ .

If they are to be **non-strict** then in an application like  $f e$  the *unevaluated* argument  $e$  is substituted into the body of  $f$ .

As an example, we will trace the evaluation of McCarthy's *91* function:

$$f x = \mathbf{if } x > 100 \mathbf{ then } x - 10 \mathbf{ else } f(f(x + 11)) \quad (7.2)$$

for argument 99 assuming strict semantics, then non-strict semantics. For emphasis, the application to be evaluated next will be picked out in blue in each case.

### Strict evaluation:

$$\begin{aligned} f(99) &= \mathbf{if } 99 > 100 \mathbf{ then } 99 - 10 \mathbf{ else } f(f(99 + 11)) \\ &= f(f(99 + 11)) \\ &= f(f(110)) \\ &= f(\mathbf{if } 110 > 100 \mathbf{ then } 110 - 10 \mathbf{ else } f(f(110 + 11))) \\ &= f(100) \\ &= \mathbf{if } 100 > 100 \mathbf{ then } 100 - 10 \mathbf{ else } f(f(100 + 11)) \\ &= f(f(100 + 11)) \\ &= f(f(111)) \\ &= f(\mathbf{if } 111 > 100 \mathbf{ then } 111 - 10 \mathbf{ else } f(f(111 + 11))) \\ &= f(101) \\ &= f(\mathbf{if } 101 > 100 \mathbf{ then } 101 - 10 \mathbf{ else } f(f(101 + 11))) \\ &= 91 \end{aligned}$$

In short,

$$f(99) \rightarrow f(f(110)) \rightarrow f(100) \rightarrow f(f(111)) \rightarrow f(101) \rightarrow 91$$

Notice that strict evaluation corresponds to the *applicative order* strategy.

### Non-strict evaluation:

$$\begin{aligned} f(99) &= \mathbf{if } 99 > 100 \mathbf{ then } 99 - 10 \mathbf{ else } f(f(99 + 11)) \\ &= f(f(99 + 11)) \\ &= \mathbf{if } f(99 + 11) > 100 \mathbf{ then } f(99 + 11) - 10 \mathbf{ else } f(f(f(99 + 11) + 11)) \\ &= \mathbf{if } [\mathbf{if } 99 + 11 > 100 \mathbf{ then } 99 + 11 - 10 \mathbf{ else } f(f(99 + 11 + 11))] > 100 \mathbf{ then } \dots \mathbf{ else } \dots \\ &= \mathbf{if } 99 + 11 - 10 > 100 \mathbf{ then } f(99 + 11) - 10 \mathbf{ else } f(f(f(99 + 11) + 11)) \\ &= f(f(f(99 + 11) + 11)) \\ &= \mathbf{if } f(f(99 + 11) + 11) > 100 \mathbf{ then } f(f(99 + 11) + 11) - 10 \mathbf{ else } f(f(f(f(99 + 11) + 11) + 11)) \\ &= \dots \end{aligned}$$

and so on. Note that non-strict evaluation corresponds to the *normal order* strategy.

### Exercise 7.2 (Tedious.)

Complete the non-strict evaluation to check that it reaches the same result.

It looks like quite an effort to defer all the computations in the non-strict trace, at least by hand. Indeed, that is the case for a computer implementation as well — managing unevaluated subexpressions as *closures* is a considerable overhead. So why bother? Here's another example:

$$f x y = \mathbf{if } x == 0 \mathbf{ then } 1 \mathbf{ else } f(x - 1) (f x y)$$

Defined over the integers, the least fixpoint solution to this equation is:

$$f x y = \mathbf{if } x \geq 0 \mathbf{ then } 1 \mathbf{ else } \perp$$

Tracing the strict evaluation strategy for  $f\ 1\ 0$ :

$$\begin{aligned}
 f\ 1\ 0 &= \mathbf{if\ 1 == 0\ then\ 1\ else\ } f\ (1 - 1)\ (f\ 1\ 0) \\
 &= f\ (1 - 1)\ (f\ 1\ 0) \\
 &= f\ 0\ (f\ 1\ 0) \\
 &= f\ 0\ (\mathbf{if\ 1 == 0\ then\ 1\ else\ } f\ (1 - 1)\ (f\ 1\ 0)) \\
 &= \dots \\
 &= f\ 0\ (f\ 0\ (f\ 1\ 0)) \\
 &= \dots
 \end{aligned}$$

which *does not terminate*. But according to the least fixpoint solution,  $f\ 1\ 0 = 1$ .

Let's trace the non-strict evaluation of the same expression:

$$\begin{aligned}
 f\ 1\ 0 &= \mathbf{if\ 1 == 0\ then\ 1\ else\ } f\ (1 - 1)\ (f\ 1\ 0) \\
 &= f\ (1 - 1)\ (f\ 1\ 0) \\
 &= f\ 0\ (f\ 1\ 0) \\
 &= \mathbf{if\ 0 == 0\ then\ 1\ else\ } f\ (0 - 1)\ (f\ 0\ (f\ 1\ 0)) \\
 &= 1
 \end{aligned}$$

terminating with the correct result.

### Definition 7.3

A computation rule that gives results that agree with the least fixpoint solution of a recursive program (including non-termination exactly where the fixpoint solution is  $\perp$ ) is called a **fixpoint computation rule**.

### Summary:

- *Normal order evaluation* (non-strict evaluation) **is a fixpoint computation rule**.
- *Applicative order evaluation* (strict evaluation) **is not a fixpoint computation rule**.