4 Overloading and Evaluation

4.1 Type classes

Some functions (such as equality) are polymorphic, but not in the same regular way as parametric polymorphism. For example

```
> data UPair a = UPair a a
```

might represent unordered pairs, for which you would want $UPair\ x\ y$ to equal $UPair\ y\ x$. The mechanism for this *ad-hoc polymorphism* in Haskell is the type class. The type of equality is

```
(==) :: Eq a => a -> a -> Bool
```

which you can read as "provided a is an Eq-type, $a \to a \to Bool$". What does it take to be an Eq-type? A definition of (==)!

The concept of an Eq-type is introduced by a class declaration:

```
class Eq a where
  (==) :: a -> a -> Bool
```

and a type such as $UPair\ a$ is put in that class by a matching instance declaration:

```
> instance Eq a => Eq (UPair a) where
>    UPair p q == UPair r s | p==r      = q==s
>                           | p==s      = q==r
>                           | otherwise = False
```

or more succinctly

```
> instance Eq a => Eq (UPair a) where
>    UPair p q == UPair r s = (p==r && q==s)||(p==s && q==r)
```

The instance requires $Eq\ a$ in order to implement the equality tests on the components.

In fact the class declaration for Eq is more like

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

so that it also provides an inequality test. However an *instance* need only provide either an implementation of `(==)` or one of `(/=)` and the equation in the *class* declaration will define the other.

Similarly the instance of the equality class for lists

```
instance Eq a => Eq [a] where
  []   ==   [] = True
  x:xs == y:ys = x == y && xs == ys
  _    ==    _ = False
```

includes equality on values of a (justified by the $Eq\ a$ constraint in the header) and a recursive call of the equality test on lists.

The underlines are dummy arguments, which need not be named because their values are not used. Names could be used, but the convention of the un-named underline is good documentation because the reader does not have to check that a named argument is in fact not used.

4.2 Evaluation

Evaluation reduces *expressions* to *values*: what is a value? One way of thinking of this is that it is (a representation of) an expression that cannot be evaluated further, often called a *normal form*. Haskell evaluates expressions by (a process equivalent to) rewriting expressions, replacing left-hand sides of equations by right-hand sides, until it reaches a normal form. There are several strategies for reducing an expression to normal form: suppose that $sq\ x = x{*}x$ then:

	Eager		**Normal order**		**Lazy**
	$sq(3+4)$		$sq(3+4)$		$sq(3+4)$
=	{addition}	=	{def of sq}	=	{def of sq}
	$sq\ 7$		$(3+4)*(3+4)$		$\mathbf{let}\ x = 3+4\ \mathbf{in}\ x*x$
=	{def of sq}	=	{addition}	=	{addition}
	$7*7$		$7*(3+4)$		$7*7$
=	{multiplication}	=	{addition}	=	{multiplication}
	49		$7*7$		49
		=	{multiplication}		
			49		

Here the costs are similar, but the order is different.

```
> inf :: Integer
> inf = 1 + inf

> const :: a -> b -> a
> const x y = x
```

where *const* is standard: $const\ x\ y = x$.

$$
\begin{array}{ll}
 & const\ 3\ inf \\
= & \{\,\text{definition of } inf\,\} \\
 & const\ 3\ (1 + inf) \\
= & \{\,\text{definition of } inf\,\} \\
 & const\ 3\ (1 + (1 + inf)) \\
= & \ldots
\end{array}
\qquad
\begin{array}{ll}
 & const\ 3\ inf \\
= & \{\,\text{definition of } const\,\} \\
 & \textbf{let}\ \{\ x = 3; y = inf\ \}\ \textbf{in}\ x \\
= & 3
\end{array}
$$

Here eager evaluation never terminates!

Eager evaluation may fail to find a normal form. If there is a normal form, normal order reduction will find it, but might be expensive. Lazy evaluation is as safe as normal order, but is usually less expensive because it tries to avoid duplicating work.

4.3 Recursion

This function calculates a factorial:

```
> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

That is, $fact\ n = n!$, at least for $n \geqslant 0$. Why is this?

Provided $n > 0$, the RHS is $n \times fact\ (n-1) = n \times (n-1)! = n!$, and if $n = 0$ the RHS is $1 = 0!$. This is effectively an induction proof in which the inductive hypothesis is that the recursive calls on the RHS satisfy the *invariant* that $fact\ n = n!$.

Why is this recursion safe, whereas that for *inf* is not? Because the recursive calls are all *smaller* in the sense that we can find a *variant* which is smaller (by a fixed amount) for all recursive calls, and bounded below: in this case the argument n will do.

Of course it is only safe for (whole numbers) $n \geqslant 0$.

4.4 Non-termination

What is value of an expression like $1\ `div`\ 0$? What is the value of an expression like $length\ [0\,..\,]$? The interpreter gives it no value, but when we are doing mathematics it is useful to have a value (so that all functions are total). By definition this is a special value $\bot$ (pronounced "*bottom*"), or rather there are values $\bot$ of every type. We identify all kinds of error with this value.

So every constructed type has an additional $\bot$ value distinct from the defined values. There are three Boolean values, and ten pairs of Booleans: the nine

actual pairs in which the components take each of three values, plus an entirely undefined value.

There are many partially defined lists such as $\bot : \bot : []$ which is a list of exactly two unknown things, and $1 : 2 : \bot$ which is a list of at least two things, the first two of which are known.

Haskell can not be expected to 'produce' a bottom when given an expression whose value is $\bot$. Sometimes it will produce an error message (for example the predefined value *undefined* does just that), sometimes it will remain silent for a very long time, sometimes the Haskell interpreter will crash... in principle, anything can happen.

Notice that you cannot expect to test for $\bot$ in a program. It is provably impossible to decide mechanically whether an abitrary computation will terminate. The equation $f\ x = (x \neq \bot)$ makes sense as a mathematical definition, but

```
> f x = x /= bottom where bottom = bottom
```

does not compute this function!

4.5 Strict functions

Some functions always demand the value of their arguments, even with normal order reduction. For example arithmetic functions like $(+)$ need both their arguments to be evaluated: $0 + \mathit{undefined}$ and $\mathit{undefined} + 1$ are both undefined; however *const undefined* 1 is undefined, but *const* 0 *undefined* is zero.

A function f is *strict* if $f\ \bot = \bot$, so $(+)$ is strict in both arguments, and *const* is strict in its first argument but not in its second. Eager evaluation would make all functions strict, so normal order (and lazy) evaluation is more faithful to the mathematical intention. Crucially, constructors are not strict, and in particular $(:)$ is neither strict in the first element of the list nor in the rest of the list. This allows lazy computations to produce infinite lists a little bit at a time.

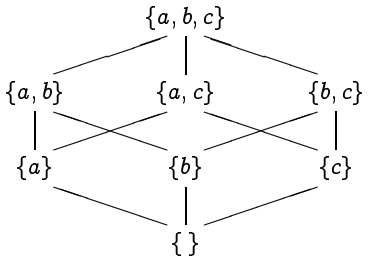
Strictness is a way of checking in the mathematics whether a value is needed in the computation. A function which uses its argument is necessarily strict in that argument (although a function can be needlessly strict in an argument which it otherwise ignores). A good compiler might analyse the strictness of functions to decide whether the value of an argument is going to be needed, and use the more efficient eager strategy on that argument.

4.6 Information ordering and computable functions

This is not on the undergradate syllabus, but may be enlightening.

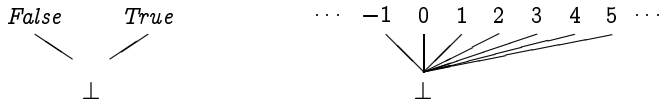
Recall that a *partial order* is a relation which, like the subset ordering on the subsets of a set, satisfies three conditions: for every x, $x \subseteq x$; for every x and y

if $x \subseteq y$ and $y \subseteq x$ then $x = y$; and for every x and y and z if $x \subseteq y$ and $y \subseteq z$ then $x \subseteq z$.

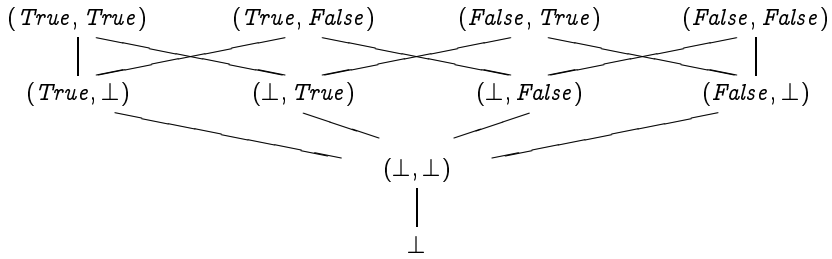


Every finite partial order can be drawn as a *Hasse diagram* in which the explicit upward edges represent just those pairs that are directly related with no value in between, and other related pairs are shown by paths up the diagram.

Informally, we can order partially evaluated expressions by how much information they yield, $x \sqsubseteq y$ if x is less useful than y but might turn into y if we did a bit more evaluation. In this ordering, $\bot \sqsubseteq y$ for all y, but for example if x and y are distinct defined integers $x \not\sqsubseteq y$ and $x \not\sqsupseteq y$.



The ten pairs of Booleans are ordered like this:



This Hasse diagram has a line going upwards between two pairs if the lower one is less well-defined than the upper one, so for example

$$\bot \sqsubseteq (\bot, \bot) \sqsubseteq (\bot, \mathit{False}) \sqsubseteq (\mathit{True}, \mathit{False})$$

and because $\sqsubseteq$ is transitive, $(\bot, \bot) \sqsubseteq (\mathit{True}, \mathit{False})$ and so on.

All computable functions are necessarily monotonic with respect to this ordering: if $x \sqsubseteq y$ then $fx \sqsubseteq fy$. It is the nature of computation that you cannot learn less about the result by supplying more information about an argument.

The monotonic functions in $\mathit{Bool} \to \mathit{Bool}$ are all strict, except for two constant functions $tt\ b = \mathit{True}$ and $ff\ b = \mathit{False}$. A (mathematical) test for equality

or inequality with bottom is not monotonic, so cannot be computable. The computable function defined by

```
> f :: Bool -> Bool
> f x = x /= bottom where bottom = bottom
```

is in fact the constant function $f\ x = \bot$.

Exercises

4.1 Show that if f and g are strict, so is the composition $f \cdot g$.
Is the converse true: that if $f \cdot g$ is strict, so must f and g be?

4.2 Which of these equations are badly typed? For the others, what can you say about the type of xs, and whether and when the equation holds?

a) $[\,] : xs = xs$	e) $xs : [\,] = [xs]$	i) $[[\,]] +\!\!+ xs = xs$
b) $[[\,]] +\!\!+ [xs] = [[\,], xs]$	f) $[\,] : xs = [[\,], xs]$	j) $xs : xs = [xs, xs]$
c) $[[\,]] +\!\!+ xs = [xs]$	g) $[xs] +\!\!+ [\,] = [xs]$	k) $xs : [\,] = xs$
d) $xs : [xs] = [xs, xs]$	h) $[[\,]] +\!\!+ xs = [[\,], xs]$	l) $[xs] +\!\!+ [xs] = [xs, xs]$

4.3 Suppose that the class of ordered types is declared by something like

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  x <  y = not (x >= y)
  x >  y = not (x <= y)
  x >= y = x == y || x > y
```

(It includes a couple of other things, and these are not quite the default definitions.) Lists are lexicographically ordered, like the words of a dictionary. Write an instance declaration for `Ord [a]`.

4.4 Time (as Richard Bird might say) for a song:

```
One man went to mow
Went to mow a meadow
One man and his dog
Went to mow a meadow

Two men went to mow
Went to mow a meadow
Two men, one man and his dog
Went to mow a meadow

Three men went to mow
Went to mow a meadow
Three men, two men, one man and his dog
Went to mow a meadow
```

Write a Haskell function $song :: Int \to String$ so that $song\ n$ is the song when there are n men (and a dog). Assume $n \leqslant 10$. To print the song, type for example: `putStr (song 5)`. You may want to start from

```
> song 0  = ""
> song n  = song (n-1) ++ "\n" ++ verse n
> verse n = line1 n ++ line ++ line3 n ++ line
```

4.5 If we count $\bot :: Bool$ as well as the proper values, there are three values of type $Bool$. So how many functions are there of type $Bool \to Bool$? How many of these are computable? Are all the computable ones definable in Haskell?

4.6 By evaluating expressions like `False && undefined` and others involving *True*, *False*, and *undefined* find exactly which function `(&&)` is implemented in the standard prelude. Give a definition which would produce that behaviour.

There is exactly one computable function, (&&&) say, which simultaneously satisfies all three equations

$$\begin{aligned} False\ \&\&\&\ y &= False \\ x\ \&\&\&\ False &= False \\ True\ \&\&\&\ True &= True \end{aligned}$$

for all x and y, including $\bot$. Given that it is computable, explain what this function does for each possible pair of arguments, defined or undefined, and so why there is only one such function. Is it definable in Haskell?