4 Expressions and Evaluation

Given an expression, the interpreter checks that the expression is well-formed, then that it is consistently typed, and only then does it evaluate the expression and print it.

4.1 Well-formed, well-typed expressions

Well-formedness is purely syntactic: parentheses match, operators have arguments, constructs (like guarded equations) are complete. This can be checked in a time bounded by the size of the expression. Purely syntactic errors in programs are caught here.

Type safety involves some information about semantics, but crucially not the values of expressions. Types can be checked or inferred in a bounded amount of time, which depends on the size of a representation of the types involved. The Haskell type system is so constructed that this check is also finite (though by careful construction it is possible to make the types exponentially big in the size of the expression so it can be quite slow; it usually is not.) Many semantic errors can be caught as type errors.

Calculating the value of an expression, though, can take for ever: for example the value of an expression can be unboundedly large. Some errors will necessarily only turn up during evaluation.

4.2 Names and operators

Names consist of a letter, followed perhaps by some alphanumeric characters, followed perhaps by some primes (single quote characters).

If the initial letter is upper case, the name may be that of a type (like *Int*) or type constructor (like *Either*) or type class (like *Eq*), or it might be the name of constructor (like *Left* or *True*). If the initial letter is lower case the name is either that of a variable or of a type variable.

Symbols made of a sequence of one or more non-alphabetical characters (excluding a few things like brackets of various sorts) behave (mostly) like infix binary operators. Some of these are predefined in the prelude, others can be defined just like any other function

```
> x +++ y = if even x then y else -y
> x // y = 2 / (1/x + 1/y)
```

They are left-associative by default, but the language allows for declaring them otherwise (like ^ which is right-associative). Parentheses convert operators into names: $x + y = (+)\ x\ y = (x+)\ y = (+y)\ x$. Conversely, backquotes convert a name into an operator: $f\ x\ y = x\ `f`\ y$.

4.3 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}
	$sq\ 7$		$(3+4)*(3+4)$		**let** $x = 3+4$ **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: $const\ x\ y = x$.

	$const\ 3\ inf$		$const\ 3\ inf$
$=$	{definition of inf}	$=$	{definition of $const$}
	$const\ 3\ (1 + inf)$		**let** $\{\ x = 3; y = inf\ \}$ **in** x
$=$	{definition of inf}	$=$	3
	$const\ 3\ (1 + (1 + inf))$		
$=$	$\ldots$		

Here eager evaluation never terminates!

4.4 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.

4.5 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 mathmatics it is useful to have a value. By definition it 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.

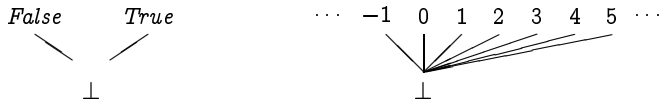
Haskell is not expected to 'produce' a bottom when given an expression whose value is $\bot$. Sometimes it will produce an error message (for example the predefined value *undefined* does just that), sometimes it will remain silent for a very long time, sometimes the Haskell interpreter will crash. .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

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 integers $x \not\sqsubseteq y$ and $x \not\sqsupseteq y$.



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

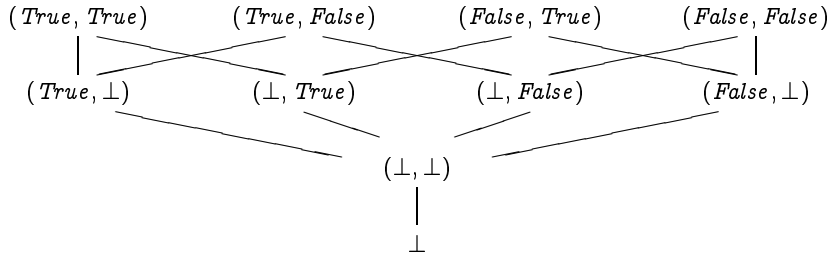
A (mathematical) test for equality or inequality with bottom is not monotonic, so cannot be computable.

4.6 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 $\mathit{const}\ \mathit{undefined}\ 1$ is undefined, but $\mathit{const}\ 0\ \mathit{undefined}$ 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 mathmatical 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.

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



and 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.

Strictness is a way of checking in the mathematics whether a value is needed. 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.