# 3 Expressions, Types and Parametric Polymorphism

Haskell programs are statically typeable, meaning that the type of an expression can be established from the components without evaluating the expression. The language is strongly typed: an expression has no value unless its type can be inferred, or checked against a claimed type. Its value has to be one of the values of its type.

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.

## 3.1 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.

## 3.2 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: $f\ x\ y = x\ \text{`}f\text{`}\ y$.

## 3.3 Types

There are some built-in types, such as *Int*, *Float*, *Char*. There are also some built-in type constructors, in particular the function type $a \to b$ for any types $a$ and $b$. The types of lists, such as [*Int*], will turn out to be built-in only in as much as there is a special syntax. Similarly, tuples such as pairs (*Int*, *Char*), triples (*Int*, *Char*, *Bool*), and so on. There are also empty tuples, () whose sole value is ().

Declarations like

```
type String = [Char]
```

introduce *type synonyms*, new *names* for existing types.

You might think that *Bool* has to be built in, but it could be (and is) defined by

```
data Bool = False | True
```

This declaration introduces a new type, *Bool* and new constants *False* and *True*. Similarly there are predefined types

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

As well as introducing a range of types, *Maybe a* for each type *a* and *Either a b* for each type *a* and type *b*, these declarations introduce

```
Nothing :: Maybe a
Just :: a -> Maybe a
Left :: a -> Either a b
Right :: b -> Either a b
```

Values of *Either Int Char* include *Left* 42 and *Right* 'x'.

A **data** declaration something akin to a record in other programming languages

```
> data PairType a b = Pair a b
```

introduces a family of types *PairType a b* for each pair of types *a* and *b*, together with a function

```
Pair :: a -> b -> PairType a b
```

This new type is equivalent to $(a, b)$.

The function *Pair*, exceptionally for the names of values in Haskell, has a name spelled with an upper case initial, which marks it out (like *True* and *False*, *Left* and *Right* and so on) as a *constructor*: a function which by the form of its definition must be invertible. Constructors can appear on the left hand side of a function definition, such as

```
> sumPair :: PairType Int Int -> Int
> sumPair (Pair x y) = x + y
```

I might usually have chosen the same name for both *Pair* and *PairType*.

## 3.4 Polymorphic types

Many standard functions have types that mark them out as being applicable to arguments of many types, for example

```
map :: (a -> b) -> [a] -> [b]
reverse :: [a] -> [a]
```

Types with lower case names are *type variables*, and these types should be read as being "for all types $a$ and $b$, *map* can have type $(a \to b) \to [a] \to [b]$", and "for all types $a$, *reverse* can have type $[a] \to [a]$".

When a polymorphic function is applied to an argument of a more constrained type (or vice versa) the polymorphic type is constrained to match.

If you want an instance of *reverse* that applies only to lists of *Char*, and cannot accidentally be applied to a $[Int]$, the expression $reverse :: [Char] \to [Char]$ will do that. (This is deliberately similar to the type signature declaration.)

This is *parametric polymorphism* and the value of a parametrically polymorphic expression is very constrained by its type. For example, there is a standard function

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Such a function has to satisfy

```
either f g (Left x)  = ...
either f g (Right y) = ...
```

where the right-hand sides are both of type $c$, whatever that type happens to be. There is in each case essentially only one possible expression known to be of that type: $f\ x$ and $g\ y$ respectively.

A parametrically polymorphic function has to operate uniformly on all possible instances of its argument type. The type of *reverse* requires that it treats all lists the same: it cannot inspect the elements of the list because it cannot know what they are. It turns out that a consequence of this is that for any $\mathit{fun} :: [a] \to [a]$

$$\mathit{map}\ f \cdot \mathit{fun} = \mathit{fun} \cdot \mathit{map}\ f$$

You can easily see that this is true for *reverse*; it is also true for any other function (and there are many) of the same type; but the force of the result is that a function that does not satisfy this equation cannot have that type.

Such results, theorems that follow from the types, were called *theorems for free* in a paper by Phil Wadler; clearly a mathematical salesman.

## 3.5 Selectors, Discriminators, Deconstructors

A constructor like *PairType* makes a value (the jargon is a *product*) which behaves like a record with two fields. These can be recovered by *selector* functions

```
> first :: PairType a b -> a
> first (Pair x y) = x
> second :: PairType a b -> b
> second (Pair x y) = y
```

The constructors of a type like an *Either* make a value (a *sum*) which behaves like a union of two possible fields. The selectors for this type are partial functions

```
> left :: Either a b -> a
> left (Left x) = x
> right :: Either a b -> b
> right (Right y) = y
```

but they are little use without a *discriminator* that tells you which kind of value you have, for example

```
> isLeft :: Either a b -> Bool
> isLeft (Left x) = True
> isLeft (Right x) = False
```

I will try to use the term *deconstructors* for the discriminator and selectors of a type. Informally, the community often refers to the discriminator and selectors for a type as the *destructors*, although this is not a good usage: it confuses them with the operations in languages that requires the programmer to manage the memory occupied by data structures in a program.

Deconstructors for a type are enough to let you write functions on that type without pattern matching, for example

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g e = if isLeft e then f (left e) else g (right e)
```

## 3.6 Recursive types

If the type being defined appears on the right hand side of a data definition, such as

```
> data List a = Nil | Cons a (List a)
```

it allows for recursive values: *Nil*, *Cons* 1 *Nil*, *Cons* 1 (*Cons* 2 *Nil*), ... are all of type *List Int*. In fact the built-in list type $[a]$ is like this, but with constructors $[\,]$ and $(:)$. The notation $[1, 2, 3]$ is shorthand for $1 : (2 : (3 : [\,]))$ (and the parentheses can be left out since $(:)$ is right-associative).

Functions over recursive types are also naturally defined by recursion. Functions over lists will often be naturally expressed by two equations: one for the empty case and one for the non-empty case, and the non-empty case will often include a recursive call of the same function.

Take for example

```
map f xs = [ f x | x <- xs ]
```

then you can calculate that

```
map f    []  = []
map f (x:xs) = f x : map f xs
```

and these two equations will do as a definition.

## 3.7 Type inference

How does Haskell check the types of expressions? The essential rule is that if $f$ is applied to $x$, then $f$ needs to have a function type $a \to b$, and $x$ needs to have type $a$, and the resulting application $f\ x$ has type $b$.

A definition such as

```
flip f x y = f y x
```

need not have an explicit type signature because Haskell can deduce a type for *flip* from this rule. The algorithm allocates a type variable to each name

$$\mathit{flip} :: \alpha \qquad f :: \beta \qquad x :: \gamma \qquad y :: \delta$$

and to the result of each application, then assembles the constraints:

$$\begin{array}{lcl} \mathit{flip}\ f & \Rightarrow & \alpha = \beta \to \epsilon \\ (\mathit{flip}\ f)\ x & \Rightarrow & \epsilon = \gamma \to \zeta \\ ((\mathit{flip}\ f)\ x)\ y & \Rightarrow & \zeta = \delta \to \eta \\ f\ y & \Rightarrow & \beta = \delta \to \theta \\ (f\ y)\ x & \Rightarrow & \theta = \gamma \to \iota \end{array}$$

and then finally because both sides are equal, $\iota = \eta$. These constraints can be met by substitution:

$$\begin{aligned} \alpha &= \beta \to \epsilon \\ &= (\delta \to \theta) \to \gamma \to \zeta \\ &= (\delta \to \gamma \to \iota) \to \gamma \to \delta \to \eta \\ &= (\delta \to \gamma \to \iota) \to \gamma \to \delta \to \iota \end{aligned}$$

which (up to changing the names of the variables) is the type with which *flip* is declared.

When you are doing type inference by hand, you need not be quite so systematic. You can anticipate some of the later substitutions because, for example, you can see that *flip* is applied to three arguments in turn, so its type must have the form $\alpha \to \beta \to \gamma \to \delta$ and you can start from there.

If you give a type declaration, it will be checked against the inferred type. Type declarations, as well as being good documentation, have the advantage of improving the explanation of any type errors found.

## Exercises

3.1 Generalising an earlier exercise: for finite types $a$, $b$ and $c$ there are as many functions of type $a \to b \to c$ as there are are of type $(a, b) \to c$ (because as numbers $(c^b)^a = c^{b \times a}$).

This correspondence, and the similar one for infinte types, is demonstrated by the (predefined) functions

```
curry   :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```

for which both $curry \cdot uncurry$ and $uncurry \cdot curry$ are identity functions (of the appropriate type).

There is a unique total function of each of these types. Write out what must be the definitions of these two functions, and prove that they are mutually inverse. (If you type these definitions at an interpreter, remember to change their names to avoid clashing with the Prelude functions.)

3.2 Suppose $h\ x\ y = f\ (g\ x\ y)$. Which of the following are true, which are false, and (in each case) why?

1. $h = f \cdot g$
2. $h\ x = f \cdot g\ x$
3. $h\ x\ y = (f \cdot g)\ x\ y$

3.3 Give most general types for the following, where possible

```
> subst f g x = (f x) (g x)
> fix f = f (fix f)
> twice f = f . f
> selfie f = f f
```

You should try to work out what the most general type is by hand, but you can check that you are right by using an interpreter; and if you are wrong, check that you understand why.

3.4 Give most general types for the following, where possible

```
> const x y = x
> comp x y z = x (y z)
> flip x y z = x z y
> dup x y = x y y
```

Deduce the types of each of

*dup const*
*comp* (*comp* (*comp dup*) *flip*) (*comp comp*)
*comp* (*comp dup*) (*comp comp flip*)

You should try to work out what the most general type is by hand, but you can check that you are right by using an interpreter; and if you are wrong, check that you understand why.

3.5 Suppose that $f\ x = (x, x)$. What is the most general type of $f$, and what is the most general type of $f \cdot f$? Use $f$ to explain how to construct a family of expressions, $E_n$, where the number of symbols in $E_n$ is no more than $n$ but there are at least $2^n$ symbols in the type expression for the type of $E_n$.

3.6 Let $\mathit{fix}\ f = f(\mathit{fix}\ f)$. What are the types of

$\mathit{fix}$
$\mathit{fix} \cdot \mathit{fix}$
$\mathit{fix} \cdot \mathit{fix} \cdot \mathit{fix}$

Show how to construct a family of expressions $E_n$ whose size is $\mathrm{O}(n)$, but whose types have expressions whose size is $\mathrm{O}(2^n)$.

$\textbf{let}\ f = \mathit{fix}\ \textbf{in}\ f \cdot f$
$\textbf{let}\ \{\ f = \mathit{fix};\ g = f \cdot f\ \}\ \textbf{in}\ g \cdot g$

Show how to construct a family of expressions $E'_n$ whose size is $\mathrm{O}(n)$, but whose types have expressions whose size is $\mathrm{O}(2^{2^n})$.