

## Making lazy functions strict

Argument evaluation can be forced by using the operator `($!)`, whose meaning (but *not* its implementation) is described by the equation:

$$f \$! x = \text{if } x \neq \perp \text{ then } f x \text{ else } \perp$$

This has the effect of evaluating `x` and then applying `f` to the result. In other words, *lazy* evaluation is replaced by *innermost* evaluation.

Provided  $x \neq \perp$ ,  $f \$! x = f x$ .

When using `($!)`, note that

- `($!)` is an operator, so it is used in expressions, not in definitions' left-hand sides
- `($!)` associates to the *right*

Example:

```
Prelude> False && (3/0 == 1)
False
```

```
Prelude> (False &&) (3/0 == 1)
False
```

```
Prelude> (False &&) $! (3/0 == 1)
Program error: {primDivDouble 3.0 0.0}
```

A *strictified* version of `fact` (Slide 60)

```
> factS n a
> = if n==0 then a else factS (n-1) $! (n*a)
```

A calculation:

```
factS 4 1
~~> factS 3 4
~~> factS 2 12
~~> factS 1 24
~~> factS 0 24
~~> 24
```

## Types

Despite the absence of type declarations in all of our examples so far, Haskell is *statically typed*: types are checked at compile time.

In most cases, the compiler can infer types without declarations. Nevertheless, in this class we **require** that Haskell definitions be accompanied by type declarations.

For example, adding

```
> fac2 :: Int -> Int
```

to the script in which `fac2` is defined would declare `fac2` to be applicable to `Int` arguments, returning `Int` results.

Type declarations have the form

$$\text{Expression} :: \text{TypeExp}$$

and the (partial) syntax of type expressions is

$$\text{TypeExp} ::= (\text{TypeName} \mid \text{TypeVar}) [ \rightarrow \text{TypeExp} ]$$

$$\text{TypeName} ::= \text{Uppercase} \{ \text{Letter} \}$$

$$\text{TypeVar} ::= \text{Lowercase} \{ \text{Letter} \}$$

A function's type **always** includes not only the type of its result, but also the types of its argument(s). Each argument's type is followed by `"->"`.

So

$$\backslash x y \rightarrow \text{if } x \text{ then } y+2 \text{ else } y-3 :: \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int}$$

Compare with C++:

```
f :: Int -> Int -> Int      -- Haskell
int f( int, int );        // C, C++, Java
```

## Polymorphic types

When the type of an argument or result is unconstrained by how it is used, it is represented in a type expression by a **type variable**.

For example, because  $(\backslash x \rightarrow 7)$  can be applied to arguments of *any* type whatsoever, with no restrictions, its type is

$$(\backslash x \rightarrow 7) :: a \rightarrow \text{Int}$$

A type whose type expression contains a variable is known as a *polymorphic* type.

Type variables follow the usual rules for variables in expressions, so

$$\backslash x \rightarrow x :: a \rightarrow a$$

means that

- $(\backslash x \rightarrow x)$  (the *identity function*) can be applied to arguments of any type whatsoever
- the result's type is the same as the argument's type.

A type variable's scope is limited to the type expression in which it appears.

## Restricted polymorphism

Some functions are applicable to arguments of more than one type, but not of all types. For example,

$$\backslash x \rightarrow x + 2 :: \text{Num } a \Rightarrow a \rightarrow a$$

because the types

- for which operator `(+)` is defined
- which can be represented by sequences of digits are exactly the types belonging to the *type class* `Num`, which contains `Int`, `Integer`, `Float`, and `Double`.

Both `(+)` and `2` are said to be *overloaded*.

## Hugs gives expressions' types

HUGS98 can be asked to show any expression's type.

The show-type command is `:type` (or just `:t`).

For example

```
Prelude> :t (+)
```

```
(+) :: Num a => a -> a -> a
```

```
Prelude> :t 2
```

```
2 :: Num a => a
```

```
Prelude> :t \x y -> if x then y+2 else y-3
```

```
\x y -> if x then y+2 else y-3 :: Num a => Bool -> a -> a
```

The last type is not the same as the one given above for the same function:

$$\backslash x y \rightarrow \text{if } x \text{ then } y+2 \text{ else } y-3 :: \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int}$$

but both are correct.

The type given by HUGS98 is the function's *most general* type. The compiler checks any item's declared type for consistency with its most general type.

It's OK to declare an item to have a less general type provided it's consistent with the item's most general type, i.e., it can be obtained from the most general type by replacing variables with non-variables. The effect is to restrict the function's type to the declared one.

The difference between `Int` and `Integer` is illustrated by the following examples:

```
Main> fac2 (20::Int)
-2102132736

Main> fac2 (20::Integer)
2432902008176640000
```

These examples exhibit several other aspects of Haskell's type system:

- `Int` consists of the limited-magnitude integers furnished by the processor.
- `Integer` is a type of unlimited-magnitude integers.
- An overloaded item can be restricted to a specific type by giving the restricted type explicitly.
- As in most programming languages, many built-in functions, e.g. `(*)`, are overloaded.
- Unlike most programming languages, Haskell functions which apply overloaded functions to their arguments, e.g. `fac2`, "inherit" the overloading.

Haskell's other main "built-in" types

```
Float, Double -- the usual floating-point values
Bool          -- {False, True}
Char          -- ascii
String       -- list of Char
```

## Numeric type conversion

In Haskell (unlike C/C++), type conversion is not automatic.

Numeric conversion functions are provided, but they must be applied explicitly.

Each numeric type `b` provides functions

```
fromInteger :: Integer -> b
fromInt     :: Int    -> b
```

In addition, `Float` and `Double` provide

```
truncate, round :: a -> b
```

```
where a ∈ { Float, Double }
      b ∈ { Int, Integer }
```

Note: The selection of the result type (`Int` or `Integer`) depends on the context.

Integer literals are *overloaded*:

"An integer numeral (without a decimal point) is ... equivalent to an application of `fromInteger` to the value of the numeral as an `Integer`."

```
Prelude> 3.0 + 4
7.0
```

```
Prelude> 3.0 + (2+4)
9.0
```

```
Prelude> 3.0 + (6 :: Int)
ERROR - Illegal Haskell 98 class constraint in
inferred type
*** Expression : 3.0 + 6
*** Type      : Fractional Int => Int
```

```
Prelude> :t 3
3 :: Num a => a
```

## Higher-order functions

A Haskell function can

- take functional arguments
- return functional results

From such functions—called *higher-order* functions—comes much of functional programming's power.

Example: `flip` applies a given function to two arguments in reverse order:

```
> flip :: (a -> b -> c) -> (b -> a -> c)
> flip f x y = f y x
> subtract :: Int -> Int -> Int
> subtract = flip (-)
```

Then

```
subtract 5 9
~> flip (-) 5 9
~> (-) 9 5
~> 9 - 5
```

More useful is function composition `(.)`:

```
> (.) :: (b -> c) -> (a -> b) -> (a -> c)
> (f . g) x = f (g x)
```

Example:

```
> even n = n `rem` 2 == 0
> odd   = not . even
```

Then

```
odd y
~> (not . even) y
~> not (even y)
```

## Aggregate types

Aggregate (composite) types are useful for making values which are collections of simpler values.

Haskell has two built-in kinds of aggregate types (actually, type constructors) and provisions for defining new ones.

The built-in type constructors are

- **tuples**, for heterogeneous collections of fixed length
- **lists**, for homogeneous collections of arbitrary length

## Tuples

We often want a function to return several values simultaneously. That's not actually possible in Haskell, but we can get the same effect using tuples, which package several values into one.

Example:

```
> quotAndRem :: Int -> Int -> (Int,Int)
> quotAndRem n d = (n `div` d, n `rem` d)
Main> quotAndRem 17 4
(4,1)
```

A tuple is built using parentheses and commas; its type is the tuple of its components' types.

The parenthesis/comma syntax appears in two rôles:

- **value constructor**, as in `(3,True,"True")`
- **type constructor**, as in `(Int,Bool,String)`