

Type Checking

- Strongly typed languages such as Java give programmers security by restricting their freedom to make mistakes
- Weakly typed languages such as Prolog allow programmers more freedom and flexibility with a reduced amount of security
- The Haskell type system gives the security of strong type checking as well as flexibility
- Every expression must have a valid type, but it is not necessary for the programmer to include type information; this is inferred automatically

Type Checking

- Given little or no explicit type information, Haskell can infer all the types associated with a function definition
- For example:
 - the types of constants can be inferred automatically
 - identifiers must have the same type everywhere within an expression
- Other type checking rules are applied for each form of expression
- For example, consider definitions with guards:

```
function pattern
  | guard1 = e1
  | guard2 = e2
  | ...
  | guardn = en
```

- The following type checking rules apply:
 - each of the guards g_i must be of type `Bool`
 - the values e_i must have the same type

Type Checking

- Consider the following function:
`isSpace c = c == ' '`
- The constant `' '` is a character
- Because `c` is compared to a `Char`, it must also be a `Char`
- The argument to `isSpace` must therefore be a `Char`
- The body of `isSpace` is a comparison, so it must be of type `Bool`
- The return type of `isSpace` is therefore a `Bool`
- `isSpace` is therefore of type `Char -> Bool`
- We can use Hugs to query the type of this function as follows:
`:type isSpace`
- Hugs will respond as follows:
`isSpace :: Char -> Bool`

Type Checking

- Type inference leaves some types completely unconstrained; in such cases the definition is *polymorphic* (this is *parametric polymorphism*)
- For example, consider the function `reverse` for reversing a list
- The following type is inferred for this function, meaning `reverse` has a *set* of types:
`[a] -> [a]`
- When we apply `reverse`, Haskell works out which type `reverse` is being used at:
 - `reverse [1,2,3]`: the function is applied to a list of `Int` and so it is used at type `[Int] -> [Int]`
 - `reverse ['a','b','c']`: the function is applied to a list of `Char` and so it is used at type `[Char] -> [Char]`

Type Checking

- In the previous example $[a] \rightarrow [a]$, a is a *type variable*
- Types with variables (eg. a , b) are called *polytypes*
- Types without type variables are called *monotypes* (eg. $[Int]$, $[Bool]$)
- A polytype can be thought of as standing for a collection of monotypes which can be obtained by instantiating the type variable with monotypes (eg. $[Int]$, $[Bool]$)

Type Checking

- Hugs can work out the type of any expression and any function from its definition
- Even if you declare the type of a function, Hugs will work it out anyway and check whether you are right
- You should *always* declare the type of any functions you are defining in your programmes:
 - the type of a function is a basic part of its design; how can you be clear about a function's behaviour unless you at least know the types of its arguments and the type of its result?
 - type declarations are an important part of programme documentation
 - type declarations help you to find errors
- If Hugs figures out that the type of a function is different from to what you expect, then you have made an error

Type Checking

- For example, suppose you wish to design a function `myEven` such that `myEven x` return `True` if `x` is divisible by 2 and `False` otherwise (there is already a standard function `even`, hence the name)

- How about:

```
myEven :: Int -> Bool
myEven x = x `div` 2
```

- When loading a script containing this definition, Hugs will give an error message:

```
Type checking
ERROR . . . : Type error in function binding
*** term : myEven
*** term : Int -> Int
*** does not match : Int -> Bool
```

- Hugs is indicating that it can tell from the definition that `myEven` has a type that is different from its declaration
- If we had not declared its type, a less obvious error message would have been sent when we used `myEven`