13.5 Type checking and type inference: an overview

Now that we have covered classes, we can see that every value in Haskell has a defined type, which might be monomorphic, polymorphic, or involve one or more type class constraints in a context. For example,

```
'w'    :: Char
flip   :: (a -> b -> c) -> (b -> a -> c)
elem   :: Eq a => a -> [a] -> Bool
```

Strong typing means that we can **check** whether or not expressions we wish to evaluate or definitions we wish to use obey the typing rules of the language, without any evaluation taking place. The benefit of this is obvious: we can catch a whole lot of errors before we run a program.

Type declarations or type inference?

Haskell types can be *inferred* from expressions and definitions, and so it is possible never to write a type declaration. For example, we can write a definition like this:

```
prodFun f g = \x -> (f x, g x)
```

either in a module or directly in GHCi, and then ask for its type in CHGi like this:

```
*TypeError> :type prodFun
prodFun :: (t -> t1) -> (t -> t2) -> t -> (t1,t2)
```

Because of this facility, some Haskellers never write a type declaration, but others, including the author, always do: why?

- The type of an object is the *most important single piece of documentation* for the object, since it tells us how it can be used – what arguments need to be passed to it, and what type the result has – without us having to understand precisely how it is implemented.
- We can use a type declaration to *give a more specific type to a definition*. This was the mechanism underlying the first part of the book, which turned polymorphic functions into monomorphic versions. To be clear, if we define `prodFun` like this

  ```
  prodFun :: (Int -> Bool) -> (Int -> Char) -> Int -> (Bool,Char)
  prodFun f g = \x -> (f x, g x)
  ```

  then it will have this more specific type. It is not difficult to recover the most general type for the definition: just comment out the type declaration.
- In writing a type declaration we are saying *what type we think a function has*. We may have not got this right, and the function is properly typed, but has a different type. For instance, typing

  ```
  fun :: Int -> Bool -> Int

  fun True 0 = 0
  fun True n = n-1
  fun _ n    = n
  ```

gives rise to this error in GHCi:

```
Couldn't match expected type 'Int' against inferred type 'Bool'
In the pattern: True
In the definition of 'fun': fun True 0 = 0
```

In a case like this it is useful to *know* that we were wrong, and then we can either correct the type declaration, or modify the function so that it has the type we wanted. Here the problem is fixed by swapping the types of the two arguments.

There is one case where we *do* need to use type declarations or annotations: this is in resolving ambiguity due to overloading (we talked briefly about this earlier, in Section 13.4, page 304).

Types and libraries

Types are also useful in locating functions in a library. Suppose we want to define a function to remove the duplicate elements from a list, transforming `[2,3,2,1,3,4]` to `[2,3,1,4]`, for instance. Such a function will have type

```
(Eq a) => [a] -> [a]
```

A Hoogle search of the standard prelude and libraries reveals just one function of this type, namely `nub`, which does exactly what we want. Plainly in practice there might be multiple matches (or missed matches because of the choice of parameter order) but nonetheless the types provide a valuable way into the Haskell library.

Overview

In the remainder this chapter we give an informal overview of the way in which types are checked. We start by looking at how type checking works in a monomorphic framework, in which every properly typed expression has a single type. Building on this, we then look at the polymorphic case, and see that it can be understood by looking at the **constraints** put on the type of an expression by the way that the expression is constructed. Crucial to this is the notion of **unification**, through which constraints are combined. We conclude the chapter by looking at the **contexts** which contain information about the class membership of type variables, and which thus manage **overloading**.

13.6 Monomorphic type checking

In this section we look at how type checking works in a monomorphic setting, without polymorphism or overloading. The main focus here is type-checking function applications. The simplified picture we see here prepares us for Haskell type checking in general, which is examined in the section after this.

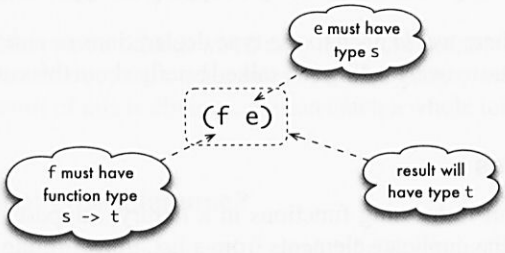
We look first at the way that we type-check expressions, and then look at how definitions are type-checked.

Expressions

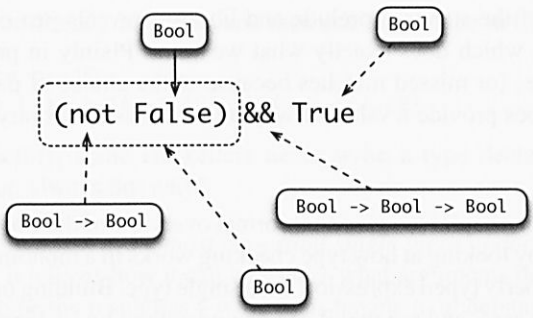
In general, an expression is either a literal, a variable or a constant or it is built up by applying a function to some arguments, which are themselves expressions.

The case of function applications includes rather more than we might at first expect. For example, we can see list expressions like `[True,False]` as the result of applying the constructor function, `':'`, thus: `True:[False]`. Also, operators and the `if ... then ... else` construct act in exactly the same way as functions, albeit with a different syntax.

The rule for type checking a function application is set out in the following diagram, where we see that a function of type `s -> t` must be applied to an argument of type `s`. A properly typed application results in an expression of type `t`.

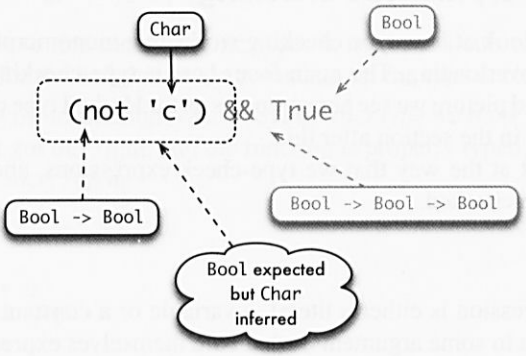


We now look at two examples. First we take `(not False) && True`, a correctly typed expression of type `Bool`,



The application of `not` to `False` results in an expression of type `Bool`. The second argument to `&&` is also a `Bool`, so the application of `&&` is correctly typed, and gives a result of type `Bool`.

If we modify the example to `(not 'c') && True`, we now see a type error, since a character argument, `'c'`, is presented to an operator expecting a `Bool` argument, `not`.



The GHCi error message for this indicates the cause of the problem:

```
Couldn't match expected type 'Bool' against inferred type 'Char'
In the first argument of 'not', namely 'c'
In the first argument of '(&&)', namely '(not 'c')'
In the expression: (not 'c') && True
```

Function definitions

In type-checking a monomorphic function definition such as

```
f :: t1 -> t2 -> ...  -> tk -> t                    (fdef)

f p1 p2 ··· pk
  | g1      = e1
  | g2      = e2
  ...
  | gl      = el
```

we need to check three things.

- Each of the guards g_i must be of type `Bool`.
- The value e_i returned in each clause must be of type `t`.
- The pattern p_j must be consistent with type of that argument, namely t_j.

A pattern is **consistent** with a type if it will match (some) elements of the type. We now look at the various cases. A variable is consistent with any type; a literal is consistent with its type. A pattern `(p:q)` is consistent with the type `[t]` if `p` is consistent with `t` and `q` is consistent with `[t]`. For example, `(0:xs)` is consistent with the type `[Int]`, and `(x:xs)` is consistent with any type of lists. The other cases of the definition are similar.

This concludes our discussion of type checking in the monomorphic case; we turn to polymorphism next.

Exercises

13.17 Predict the type errors you would obtain by defining the following functions:

```
f n     = 37+n
f True  = 34

g 0 = 37
g n = True

h x
  | x>0          = True
  | otherwise    = 37

k x = 34
k 0 = 35
```

Check your answers by typing each definition into a Haskell script, and loading the script into GHCi. Remember that you can use `:type` to give the type of an expression.

13.7 Polymorphic type checking

In a monomorphic situation, an expression is either well typed, and has a single type, or is not well typed and has none. In a polymorphic language like Haskell, the situation is more complicated, since a polymorphic object is precisely one which has many types.

In this section we first re-examine what is meant by polymorphism, before explaining type checking by means of **constraint satisfaction**. Central to this is the notion of **unification**, by which we find the types simultaneously satisfying two type constraints.

Polymorphism

We are familiar with functions like

```
length :: [a] -> Int                                    (length)
```

whose types are polymorphic, but how should we understand the type variable `a` in this type? We can see `(length)` as shorthand for saying that `length` has a **set** of types,

```
[Int] -> Int
[(Bool,Char)] -> Int
 ...
```

in fact containing all the types `[t] -> Int` where `t` is a **monotype**, that is a type not containing type variables.

When we apply `length` we need to determine at which of these types `length` is being used. For example, when we write

```
length ['c','d']
```

we can see that `length` is being applied to a list of `Char`, and so we are using `length` at type `[Char] -> Int`.

Constraints

How can we explain what is going on here in general? We can see different parts of an expression as putting different **constraints** on its type. Under this interpretation, type checking becomes a matter of working out whether we can find types which meet the constraints. We have seen some informal examples of this when we discussed the types of `map` and `filter` in Section 10.2. We consider some further examples now.

Example 1

Consider the definition

```
f (x,y) = (x , ['a' .. y])
```

The argument of `f` is a pair, and we consider separately what constraints there are on the types of `x` and `y`. `x` is completely unconstrained, as it is returned as the first half of a pair. On the other hand, `y` is used within the expression `['a' .. y]`, which denotes a range within an enumerated type, starting at the character `'a'`. This forces `y` to have the type `Char`, and gives the type for `f`:

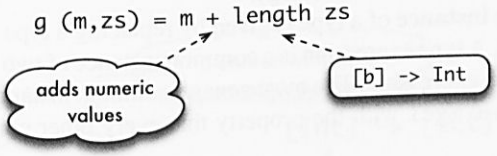
```
f :: (a,Char) -> (a,[Char])
```

Example 2

Now we examine the definition

```
g (m,zs) = m + length zs
```

What constraints are placed on the types of `m` and `zs` in this definition? We can see that `m` is added to something, so `m` must have a numeric type – which one it is remains to be seen. The other argument of the addition is `length zs`, which tells us two things.



First, we see that `zs` will have to be of type `[b]`, and also that the result is an `Int`. This forces `+` to be used at `Int`, and so forces `m` to have type `Int`, giving the result

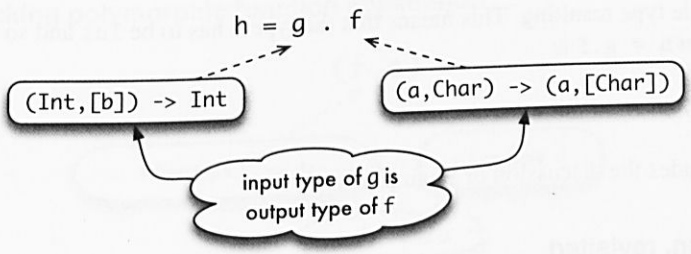
```
g :: (Int,[b]) -> Int
```

Example 3

We now consider the composition of the last two examples,

```
h = g . f
```

In a composition `g . f`, the output of `f` becomes the input of `g`,

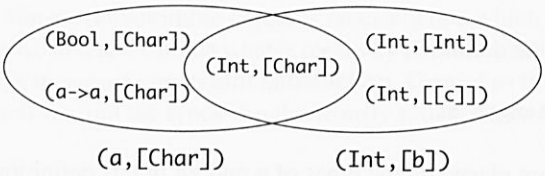


Here we should recall the meaning of types which involve type variables; we can see them as shorthand for sets of types. The output of `f` is described by `(a,[Char])`, and

the input of g by `(Int,[b])`. We therefore have to look for types which meet both these descriptions. We will now look at this general topic, returning to the example in the course of this dicussion.

Unification

How are we to describe the types which meet the two descriptions `(a,[Char])` and `(Int,[b])`?



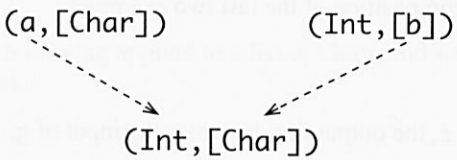
As sets of types, we look for the intersection of the sets given by `(a,[Char])` and `(Int,[b])`. How can we work out a description of this intersection? Before we do this, we revise and introduce some terminology.

Recall that an **instance** of a type is given by replacing a type variable or variables by type expressions. A type expression is a common instance of two type expressions if it is an instance of each expression. The most general common instance of two expressions is a common instance `mgci` with the property that every other common instance is an instance of `mgci`.

Now we can describe the intersection of the sets given by two type expressions. It is called the **unification** of the two, which is the **most general common instance** of the two type expressions.

Example 3 (continued)

In this example, we have



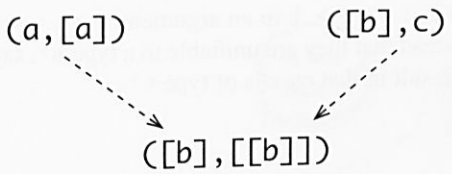
with a single type resulting. This means that the type a has to be `Int` and so the type of the function `h = g.f` is

```
h :: (Int,Char) -> Int
```

This concludes the discussion of Example 3.

Unification, revisited

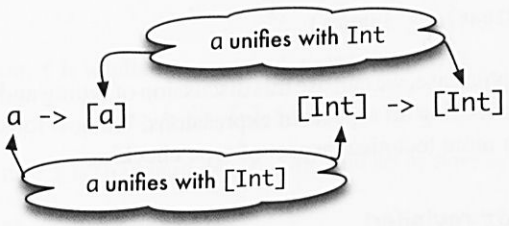
Unification need not result in a monotype. In the example of unifying the types `(a,[a])` and `([b],c)`,



the result is the type `([b],[[b]])`. This is because the expression `(a,[a])` constrains the type to have in its second component a list of elements of the first component type, while the expression `([b],c)` constrains its first component to be a list. Thus satisfying the two gives the type `([b],[[b]])`.

In the last example, note that there are many common instances of the two type expressions, including `([Bool],[[Bool]])` and `([[c]],[[[c]]])`, but neither of these examples is the unifier, since `([b],[[b]])` is not an instance of either of them. On the other hand, they are each instances of `([b],[[b]])`, as it is the most general common instance, and so the unifier of the two type expressions.

Not every pair of types can be unified: consider the case of `[Int] -> [Int]` and `a -> [a]`.

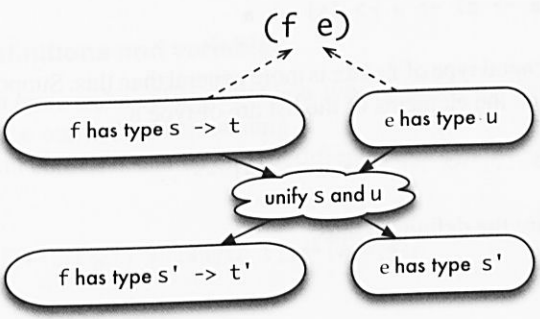


Unifying the argument types requires `a` to become `[Int]`, while unifying the result types requires `a` to become `Int`; clearly these constraints are inconsistent, and so the unification fails.

Type-checking expressions

As we saw in Section 13.6, function application is central to expression formation. This means that type checking also hinges on function applications.

Type-checking polymorphic function application



In applying a function `f :: s -> t` to an argument `e :: u` we do not require that `s` and `u` are equal, but instead that they are unifiable to a type `s'`, say, giving `e :: s'` and `f :: s' -> t'`; the result in that case is of type `t'`.

Example 4

As an example, consider the application `map Circle` where `Circle` is one of the constructor functions for the `Shape` type.

```
map :: (a -> b) -> [a] -> [b]
Circle :: Float -> Shape
```

Unifying `a -> b` and `Float -> Shape` results in `a` becoming `Float` and `b` becoming `Shape`; this gives

```
map :: (Float -> Shape) -> [Float] -> [Shape]
```

and so

```
map Circle :: [Float] -> [Shape]
```

As in the monomorphic case, we can use this discussion of typing and function application in explaining type checking all aspects of expressions. We now look at another example, before examining a more technical aspect of type checking.

Example 5, `foldr` revisited

In Section 10.3 we introduced the `foldr` function

```
foldr f s []     = s                                    (foldr.1)
foldr f s (x:xs) = f x (foldr f s xs)                   (foldr.2)
```

which could be used to fold an operator into a list, as in

```
foldr (+) 0 [2,3,1] = 2+(3+(1+0))
```

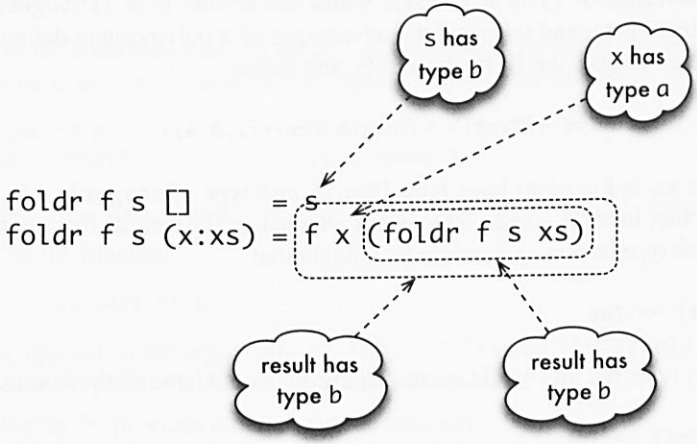
so that it appears as if `foldr` has the type given by

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

In fact, the most general type of `foldr` is more general than this. Suppose that the starting value has type `b` and the elements of the list are of type `a`

```
foldr :: (... -> ... -> ...) -> b -> [a] -> ...
```

Then we can picture the definition thus:



s is the result of the first equation, and so the result type of the `foldr` function itself will be b, the type of s

```
foldr :: (... -> ... -> ...) -> b -> [a] -> b
```

In the second equation, `f` is applied to `x` as first argument, giving

```
foldr :: (a -> ... -> ...) -> b -> [a] -> b
```

The second argument of `f` is the result of a `foldr`, and so of type b,

```
foldr :: (a -> b -> ...) -> b -> [a] -> b
```

Finally, the result of the second equation is an application of `f`; this result must have the same result type as the `foldr` itself, b.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

With this insight about the type of `foldr` we were able to see that `foldr` could be used to define another whole cohort of list functions, such as an insertion sort,

```
iSort :: Ord a => [a] -> [a]
iSort = foldr ins []
```

in which `ins` has the type `Ord a => a -> [a] -> [a]`.

Polymorphic definitions and variables

Here we examine a more technical aspect of how type checking works over polymorphic definitions; it may be omitted on first reading.

Functions and constants can be used at different types in the same expression. A simple instance is

```
expr = length ([]++[True]) + length ([]++[2,3,4])          (expr)
```

The first occurrence of `[]` is at `[Bool]`, whilst the second is at `[Integer]`. This is completely legitimate, and is one of the advantages of a polymorphic definition. Now suppose that we replace the `[]` by a variable, and define

```
funny xs = length (xs++[True]) + length (xs++[2,3,4])          (funny)
```

The variable `xs` is forced to have type `[Bool]` *and* type `[Integer]`; it is forced to be polymorphic, in other words. This is not allowed in Haskell, as there is no way of expressing the type of `funny`. It might be thought that

```
funny :: [a] -> Int
```

was a correct type, but this would mean that `funny` would have all the instance types

```
funny :: [Int] -> Int
funny :: [[Char]] -> Int
  ...
```

which it clearly does not. We conclude that constants and variables are treated differently: constants may very well appear at different incompatible types in the same expression, variables cannot.

What is the significance of disallowing the definition `(funny)` but allowing the definition `(expr)`? Taking `(expr)` first, we have a polymorphic definition of the form `[] :: [a]` and an expression in which `[]` occurs twice; the first occurrence is at `[Bool]`, the second at `[Integer]`. To allow these independent uses to occur, we type-check each use of a polymorphic definition with different type variables, so that a constraint on one use does not affect any of the others.

On the other hand, how is the definition of `(funny)` disallowed? When we type check the use of a variable we will not treat each instance as being of an independent type. Suppose we begin with no constraint on `xs`, so `xs::t`, say. The first occurrence of `xs` forces `xs::[Bool]`, the second requires `xs::[Integer]`; these two constraints cannot be satisfied simultaneously, and thus the definition `(funny)` fails to type check.

The crucial point to remember from this example is that the definition of a function can't force any of its arguments to be polymorphic.

Function definitions

In type checking a function definition like `(fdef)` on page 311 above we have to obey rules similar to the monomorphic case.

- Each of the guards g_i must be of type `Bool`.
- The value e_i returned in each clause must have a type s_i which is at least as general as `t`; that is, s_i must have `t` as an instance.
- The pattern p_j must be **consistent** with type of that argument, namely t_j.

We take up a final aspect of type checking – the impact of type classes – in the next section.

Exercises

13.18 Do the following pairs of types – listed vertically – unify? If so, give a most general unifier for them; if not, explain why they fail to unify.

```
(Int -> b)          (Int,a,a)
(a -> Bool)         (a,a,[Bool])
```

13.19 Show that we can unify `(a,[a])` with `(b,c)` to give `(Bool,[Bool])`.

13.20 Can the function

```
f :: (a,[a]) -> b
```

be applied to the arguments `(2,[3])`, `(2,[])` and `(2,[True])`; if so, what are the types of the results? Explain your answers.

13.21 Repeat the previous question for the function

```
f :: (a,[a]) -> a
```

Explain your answers.

13.22 Give the type of `f [] []` if `f` has type

```
f :: [a] -> [b] -> a -> b
```

What is the type of the function `h` given by the definition

```
h x = f x x ?
```

13.23 How can you use the Haskell system to check whether two type expressions are unifiable, and if so what is their unification? Hint: you can make dummy definitions in Haskell in which the defined value, `zircon` say, is equated with itself:

```
zircon = zircon
```

Values defined like this can be declared to have any type you wish.

13.24 [Harder] Recalling the definitions of `curry` and `uncurry` from Section 11.4, what are the types of

```
curry id
uncurry id
curry (curry id)
uncurry (uncurry id)
uncurry curry
```

Explain why the following expressions do not type-check:

```
curry uncurry
curry curry
```

13.25 [Harder] Give an *algorithm* which decides whether two type expressions are unifiable. If they are, your algorithm should return a most general unifying substitution; if not, it should give some explanation of why the unification fails.

13.8 Type checking and classes

Classes in Haskell restrict the use of some functions, such as ==, to types in the class over which they are defined, in this case Eq. These restrictions are apparent in the **contexts** which appear in some types. For instance, if we define

```
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

its type will be

```
Eq a => [a] -> a -> Bool
```

because x and y of type a are compared for equality in the definition, thus forcing the type a to belong to the equality class Eq.

This section explores the way in which type checking takes place when overloading is involved; the material is presented informally, by means of an example.

Suppose we are to apply the function member to an expression e, whose type is

```
Ord b => [[b]]
```

Informally, e is a list of lists of objects, which belong to a type which carries an ordering. In the absence of the contexts we would unify the type expressions, giving

```
member :: [[b]] -> [b] -> Bool          e :: [[b]]
```

and so giving the application member e the type [b] -> Bool. We do the same here, but we also apply the unification to the contexts, producing the context

```
(Eq [b] , Ord b)                                    (ctx.1)
```

Now, we check and simplify the context.

- The requirements in a context can only apply to type variables, so we need to eliminate requirements like Eq [b]. The only way these can be eliminated is to use the instance declarations. In this case the built-in instance declaration

  ```
  instance Eq a => Eq [a] where ....
  ```

  allows us to replace the requirement Eq [b] with Eq b in (ctx.1), giving the new context

  ```
  (Eq b , Ord b)                                    (ctx.2)
  ```

  We repeat this process until no more instances apply.

  If we fail to reduce all the requirements to ones involving a type variable, the application fails, and an error message would be generated. This happens if we apply member to [id];

  ```
  No instance for (Eq (a -> a))
    arising from a use of 'member' at <interactive>:1:0-10
  ```

```
Possible fix: add an instance declaration for (Eq (a -> a))
In the expression: member [id]
In the definition of 'it': it = member [id]
```

since `id` is a function, whose type is not it the class Eq.

- We then simplify the context using the `class` definitions. In our example we have both `Eq b` and `Ord b`, but recall that

  ```
  class Eq a => Ord a where ...
  ```

  so that any instance of `Ord` is automatically an instance of Eq; this means that we can simplify `(ctx.2)` to

  ```
  Ord b
  ```

  This is repeated until no further simplifications result.

For our example, we thus have the type

```
member e :: Ord b => [b] -> Bool
```

This three-stage process of unification, checking (with instances) and simplification is the general pattern for type checking with contexts in Haskell.

Finally, we should explain how contexts are introduced into the types of the language. They originate in types for the functions in class declarations, so that, in the example of the `Info` class from earlier in the chapter, we have

```
examples :: Info a => [a]
size     :: Info a => a -> Int
```

The type checking of functions which use these overloaded functions will propagate and combine the contexts as we have seen above.

We have seen informally how the Haskell type system accommodates type checking for the overloaded names which belong to type classes. A more thorough overview of the technical aspects of this, including a discussion of the 'monomorphism restriction' which needs to be placed on certain polymorphic bindings, is to be found in the Haskell 2010 report (Marlow 2010).

Exercises

13.26 Give the type of each of the individual conditional equations which follow, and discuss the type of the function which together they define.

```
merge (x:xs) (y:ys)
  | x<y          = x : merge xs (y:ys)
  | x==y         = x : merge xs ys
  | otherwise    = y : merge (x:xs) ys
merge (x:xs) []     = (x:xs)
merge []     (y:ys) = (y:ys)
merge []     []     = []
```

13.27 Define a polymorphic sorting function, and show how its type is derived from the type of the ordering relation

```
compare :: Ord a => a -> a -> Ordering
```

13.28 Investigate the types of the following numerical functions; you will find that the types refer to some of the built-in numeric classes.

```
mult x y = x*y
divide x = x `div` 2
share x  = x / 2.0
```

Recall that these can be given more restrictive types, such as

```
divide :: Int -> Int
```

by explicitly asserting their types as above.

Summary

This chapter has shown how names such as `read` and `show` and operators like + can be overloaded to have different definitions at different types. The mechanism which enables this is the system of Haskell classes. A `class` definition contains a signature which contains the names and types of operations which must be supplied if a type is to be a member of the class. For a particular type, the function definitions are contained in an `instance` declaration.

In giving the type of a function, or introducing a class or an instance, we can supply a context, which constrains the type variables occurring. Examples include

```
member :: Eq a => [a] -> a -> Bool
instance  Eq a => Eq [a] where ....
class     Eq a => Ord a  where ....
```

In the examples, it can be seen that `member` can only be used over types in the class `Eq`. Lists of a can be given an equality, provided that a itself can; types in the class `Ord` must already be in the class `Eq`. After giving examples of the various mechanisms, we looked at the classes in the standard preludes of Haskell.

We concluded the chapter with a discussion of how type checking of expressions and definitions is performed in Haskell, initially in the monomorphic case, and then in full generality with polymorphic and overloaded functions. In that case we saw type checking as a process of extracting and consolidating constraints which come from the unification of type expressions which contain type variables.