Chapter 13

Checking types

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.

Beyond this, types are a valuable form of program documentation: when we look at a definition, the first relevant piece of information about it is its type, since this explains how it is to be used. In the case of a function, we can read off from its type the types of values to which it has to be applied, and also the type of the result of applying it.

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 search of the standard prelude `Prelude.hs` and the library `List.hs` reveals just one function of this type, namely `nub`, which has exactly the effect we seek. 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 `handle' on the functions in a library.

In 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**.

13.1 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. When discussing polymorphic operations in this section we will use monomorphic instances, indicated by a type subscript or subscripts. For example, we write

$+_{\texttt{Int}}$ `:: Int -> Int -> Int`
$\texttt{length}_{\texttt{Char}}$ `:: [Char] -> Int`

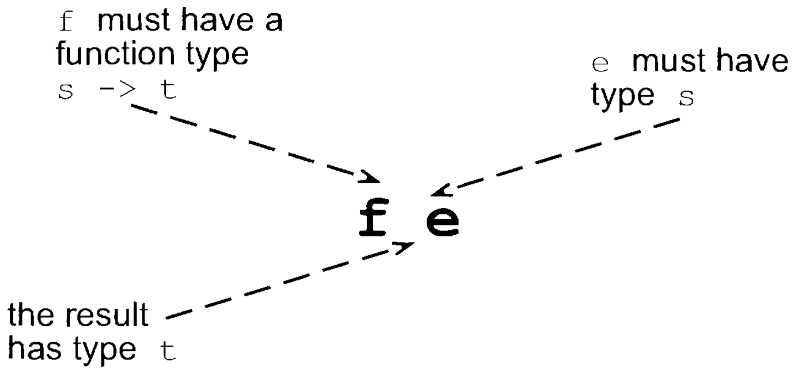
We look first at the way that we type-check expressions, and then look at how definitions are type-checked.

Expressions

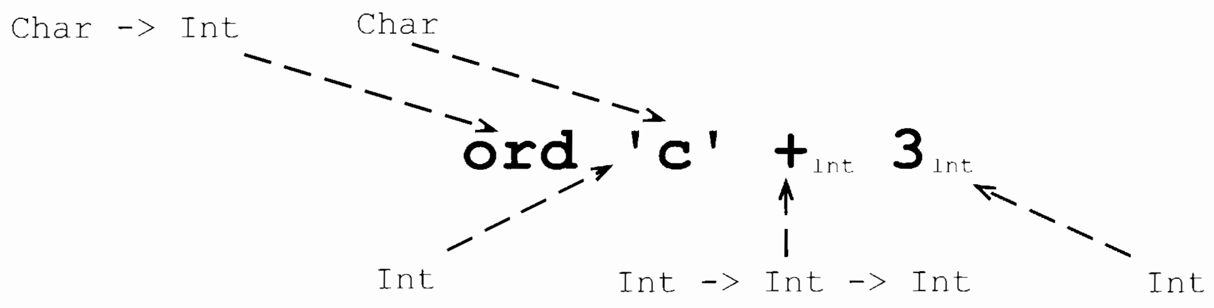
In general, an expression is either a literal, a variable or a constant or it is built up by applying a function to some arguments, which are themselves expressions.

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`.

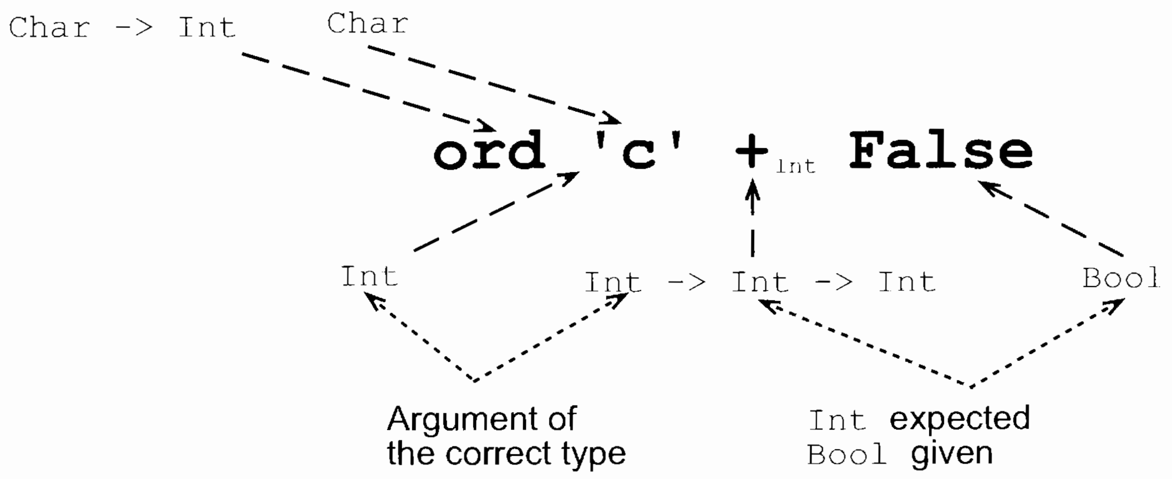


We now look at two examples. First we take ord 'c' $+_{Int}$ 3_{Int}, a correctly typed expression of type Int,



The application of ord to 'c' results in an expression of type Int. The second argument to $+_{Int}$ is also an Int, so the application of $+_{Int}$ is correctly typed, and gives a result of type Int.

If we modify the example to ord 'c' $+_{Int}$ False, we now see a type error, since a Boolean argument, False, is presented to an operator expecting Int arguments, $+_{Int}$.



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.

Exercise

13.1 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 Hugs. Remember that you can use `:type` to give the type of an expression.

13.2 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 9.2. We consider some further examples now.

Examples

1. Consider the definition

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

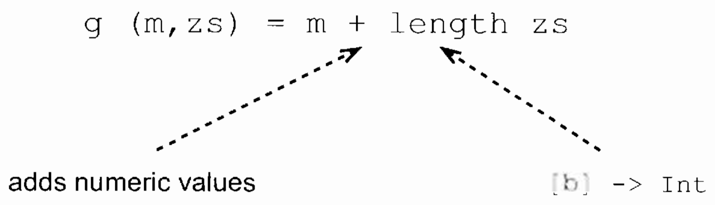
The argument of f is a pair, and we consider separately what constraints there are on the types of x and y. x is completely unconstrained, as it is returned as the first half of a pair. On the other hand, y is used within the expression ['a' .. y], which denotes a range within an enumerated type, starting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

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

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.



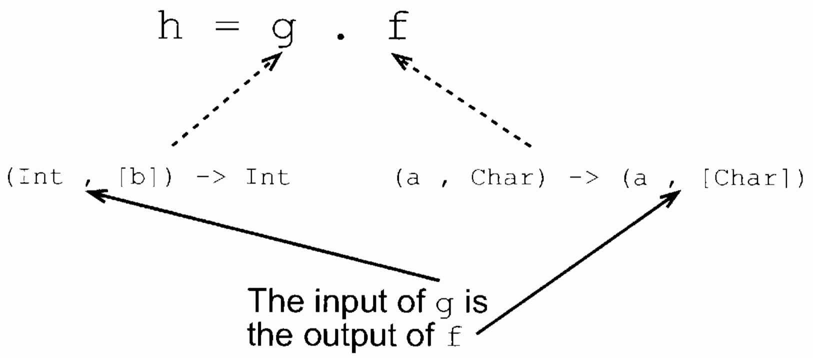
First, we see that `zs` will have to be of type `[b]`, and also that the result is an `Int`. This forces `+` to be used at `Int`, and so forces `m` to have type `Int`, giving the result

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

3. We now consider the composition of the last two examples,

```
h = g . f
```

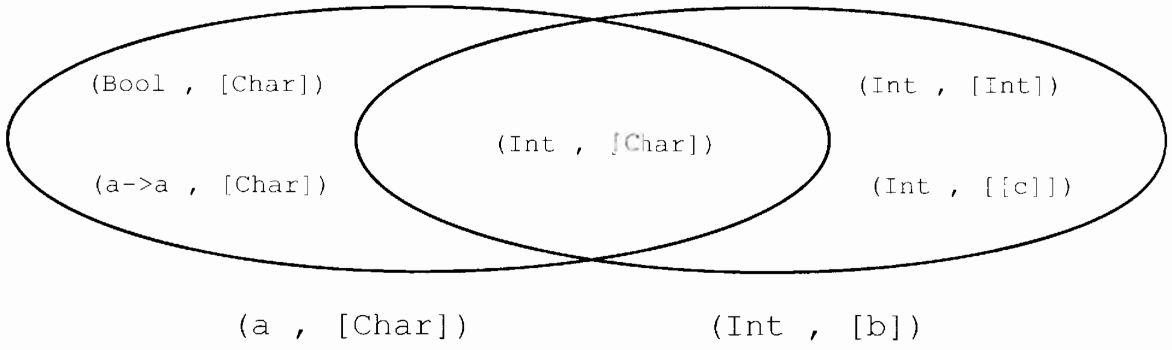
In a composition `g . f`, the output of `f` becomes the input of `g`,



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.

Examples

3 (contd) In this example, we have

$$\begin{tikzcd}
\texttt{(a , [Char])} \arrow[dr, dashed] & & \texttt{(Int , [b])} \arrow[dl, dashed] \\
& \texttt{(Int , [Char])} &
\end{tikzcd}$$

with a single type resulting. This gives the function `h` the following type

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

and this completes the discussion of example 3.

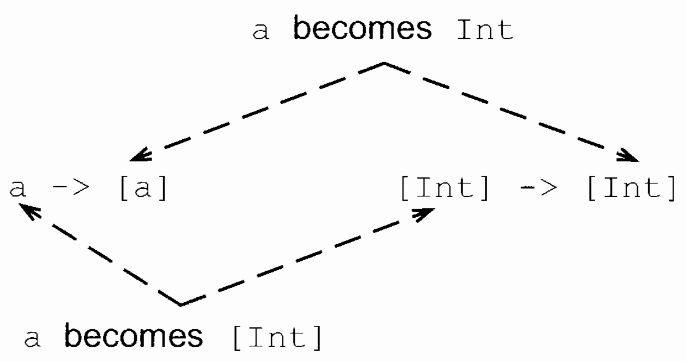
Unification need not result in a monotype. In the example of unifying the types `(a,[a])` and `([b],c)`,

$$\begin{tikzcd}
\texttt{(a , [a])} \arrow[dr, dashed] & & \texttt{([b] , c)} \arrow[dl, dashed] \\
& \texttt{([b] , [[b]])} &
\end{tikzcd}$$

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]`.

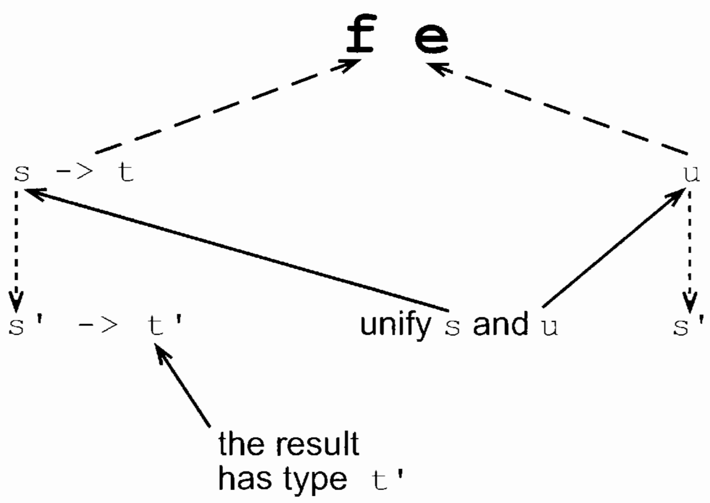


Unifying the argument types requires a to become [Int], while unifying the result types requires a to become Int; clearly these constraints are inconsistent, and so the unification fails.

Type-checking expressions

As we saw in Section 13.1, 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'. As an example, consider the application map ord where

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unifying a -> b and Char -> Int results in a becoming Char and b becoming Int; this gives

```
map :: (Char -> Int) -> [Char] -> [Int]
```

and so

```
map ord :: [Char] -> [Int]
```

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

4. `foldr` again

In Section 9.3 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))
```

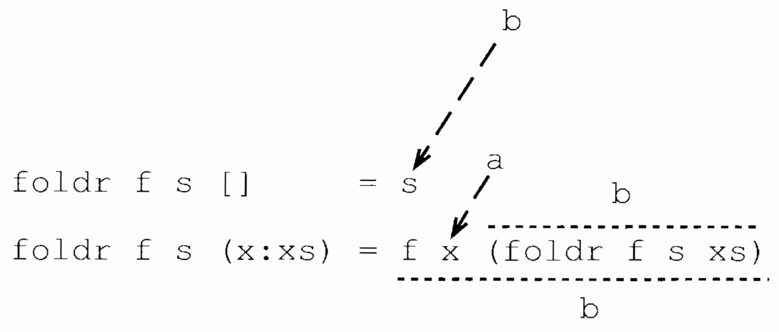
so that it appears as if `foldr` has the type given by

```
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:



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

`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
```