JFP **13** (1): 39–66, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803000613 Printed in the United Kingdom

Chapter 4

Declarations and Bindings

In this chapter, we describe the syntax and informal semantics of Haskell *declarations*.

module	$\rightarrow$	`module` *modid* [*exports*] `where` *body*	
	$\vert$	*body*	
body	$\rightarrow$	`{` *impdecls* `;` *topdecls* `}`	
	$\vert$	`{` *impdecls* `}`	
	$\vert$	`{` *topdecls* `}`	
topdecls	$\rightarrow$	$topdecl_1$ `;` ... `;` $topdecl_n$	$(n \geq 1)$
topdecl	$\rightarrow$	`type` *simpletype* `=` *type*	
	$\vert$	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
	$\vert$	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
	$\vert$	`class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]	
	$\vert$	`instance` [*scontext* `=>`] *qtycls* *inst* [`where` *idecls*]	
	$\vert$	`default` `(`$type_1$ `,` ... `,` $type_n$`)`	$(n \geq 0)$
	$\vert$	*decl*	
decls	$\rightarrow$	`{` $decl_1$ `;` ... `;` $decl_n$ `}`	$(n \geq 0)$
decl	$\rightarrow$	*gendecl*	
	$\vert$	(*funlhs* $\vert$ pat^0) *rhs*	
cdecls	$\rightarrow$	`{` $cdecl_1$ `;` ... `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	$\rightarrow$	*gendecl*	
	$\vert$	(*funlhs* $\vert$ *var*) *rhs*	

idecls	→	`{` $idecl_1$ `;` ... `;` $idecl_n$ `}`	$(n \geq 0)$
idecl	→	(*funlhs* | *var*) *rhs*	
	|		(empty)
gendecl	→	*vars* `::` [*context* `=>`] *type*	(type signature)
	|	*fixity* [*integer*] *ops*	(fixity declaration)
	|		(empty declaration)
ops	→	op_1 `,` ... `,` op_n	$(n \geq 1)$
vars	→	var_1 `,` ... `,` var_n	$(n \geq 1)$
fixity	→	`infixl` | `infixr` | `infix`	

The declarations in the syntactic category *topdecls* are only allowed at the top level of a Haskell module (see Chapter 5), whereas *decls* may be used either at the top level or in nested scopes (i.e. those within a `let` or `where` construct).

For exposition, we divide the declarations into three groups: user-defined datatypes, consisting of `type`, `newtype`, and `data` declarations (Section 4.2); type classes and overloading, consisting of `class`, `instance`, and `default` declarations (Section 4.3); and nested declarations, consisting of value bindings, type signatures, and fixity declarations (Section 4.4).

Haskell has several primitive datatypes that are "hard-wired" (such as integers and floating-point numbers), but most "built-in" datatypes are defined with normal Haskell code, using normal `type` and `data` declarations. These "built-in" datatypes are described in detail in Section 6.1.

4.1 Overview of Types and Classes

Haskell uses a traditional Hindley–Milner polymorphic type system to provide a static type semantics [5, 8], but the type system has been extended with *type classes* (or just *classes*) that provide a structured way to introduce *overloaded* functions.

A `class` declaration (Section 4.3.1) introduces a new *type class* and the overloaded operations that must be supported by any type that is an instance of that class. An `instance` declaration (Section 4.3.2) declares that a type is an *instance* of a class and includes the definitions of the overloaded operations – called *class methods* – instantiated on the named type.

For example, suppose we wish to overload the operations `(+)` and `negate` on types `Int` and `Float`. We introduce a new type class called `Num`:

```
class Num a  where          --  simplified class declaration for Num
  (+)    :: a -> a -> a     -- (Num is defined in the Prelude)
  negate :: a -> a
```

This declaration may be read "a type `a` is an instance of the class `Num` if there are class methods `(+)` and `negate`, of the given types, defined on it."

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

We may then declare `Int` and `Float` to be instances of this class:

```
instance Num Int  where     -- simplified instance of Num Int
  x + y       =  addInt x y
  negate x    =  negateInt x
instance Num Float  where   -- simplified instance of Num Float
  x + y       =  addFloat x y
  negate x    =  negateFloat x
```

where `addInt`, `negateInt`, `addFloat`, and `negateFloat` are assumed in this case to be primitive functions, but in general could be any user-defined function. The first declaration above may be read "`Int` is an instance of the class `Num` as witnessed by these definitions (i.e. class methods) for `(+)` and `negate`."

More examples of type classes can be found in the papers by Jones [10] or Wadler and Blott [16]. The term "type class" was used to describe the original Haskell 1.0 type system; "constructor class" was used to describe an extension to the original type classes. There is no longer any reason to use two different terms: in this report, "type class" includes both the original Haskell type classes and the constructor classes introduced by Jones.

4.1.1 Kinds

To ensure that they are valid, type expressions are classified into different *kinds*, which take one of two possible forms:

- The symbol $*$ represents the kind of all nullary type constructors.
- If κ_1 and κ_2 are kinds, then $\kappa_1 \rightarrow \kappa_2$ is the kind of types that take a type of kind κ_1 and return a type of kind κ_2.

Kind inference checks the validity of type expressions in a similar way that type inference checks the validity of value expressions. However, unlike types, kinds are entirely implicit and are not a visible part of the language. Kind inference is discussed in Section 4.6.

4.1.2 Syntax of Types

type	$\rightarrow$	*btype* [`->` *type*]	(function type)
btype	$\rightarrow$	[*btype*] *atype*	(type application)
atype	$\rightarrow$	*gtycon*	
	|	*tyvar*	
	|	`(` $type_1$ `,` ... `,` $type_k$ `)`	(tuple type, $k \geq 2$)
	|	`[` *type* `]`	(list type)

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

	|	`(` *type* `)`	(parenthesised constructor)
gtycon	$\rightarrow$	*qtycon*	
	|	`()`	(unit type)
	|	`[]`	(list constructor)
	|	`(->)`	(function constructor)
	|	`(,{,})`	(tupling constructors)

The syntax for Haskell type expressions is given above. Just as data values are built using data constructors, type values are built from *type constructors*. As with data constructors, the names of type constructors start with uppercase letters. Unlike data constructors, infix type constructors are not allowed (other than `(->)`).

The main forms of type expression are as follows:

1. Type variables, written as identifiers beginning with a lowercase letter. The kind of a variable is determined implicitly by the context in which it appears.

2. Type constructors. Most type constructors are written as an identifier beginning with an uppercase letter. For example:

   - `Char`, `Int`, `Integer`, `Float`, `Double` and `Bool` are type constants with kind $*$.
   - `Maybe` and `IO` are unary type constructors, and treated as types with kind $* \rightarrow *$.
   - The declarations `data T ...` or `newtype T ...` add the type constructor `T` to the type vocabulary. The kind of `T` is determined by kind inference.

   Special syntax is provided for certain built-in type constructors:

   - The *trivial type* is written as `()` and has kind $*$. It denotes the "nullary tuple" type, and has exactly one value, also written `()` (see Sections 3.9 and 6.1.5).
   - The *function type* is written as `(->)` and has kind $* \rightarrow * \rightarrow *$.
   - The *list type* is written as `[]` and has kind $* \rightarrow *$.
   - The *tuple types* are written as `(,)`, `(,,)`, and so on. Their kinds are $* \rightarrow * \rightarrow *$, $* \rightarrow * \rightarrow * \rightarrow *$, and so on.

   Use of the `(->)` and `[]` constants is described in more detail below.

3. Type application. If t_1 is a type of kind $\kappa_1 \rightarrow \kappa_2$ and t_2 is a type of kind κ_1, then $t_1\ t_2$ is a type expression of kind κ_2.

4. A *parenthesized type*, having form `(`t`)`, is identical to the type t.

For example, the type expression `IO a` can be understood as the application of a constant, `IO`, to the variable `a`. Since the `IO` type constructor has kind $* \rightarrow *$, it follows that both the variable `a` and the whole expression, `IO a`, must have kind $*$. In general, a process of *kind inference* (see

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

Section 4.6) is needed to determine appropriate kinds for user-defined datatypes, type synonyms, and classes.

Special syntax is provided to allow certain type expressions to be written in a more traditional style:

1. A *function type* has the form t_1 `->` t_2, which is equivalent to the type `(->)` t_1 t_2. Function arrows associate to the right. For example, `Int -> Int -> Float` means `Int -> (Int -> Float)`.
2. A *tuple type* has the form `(`t_1 `,` ... `,` t_k`)` where $k \geq 2$, which is equivalent to the type `(,...,)` t_1 ... t_k where there are $k-1$ commas between the parenthesis. It denotes the type of k-tuples with the first component of type t_1, the second component of type t_2, and so on (see Sections 3.8 and 6.1.4).
3. A *list type* has the form `[`t`]`, which is equivalent to the type `[]` t. It denotes the type of lists with elements of type t (see Sections 3.7 and 6.1.3).

These special syntactic forms always denote the built-in type constructors for functions, tuples, and lists, regardless of what is in scope. In a similar way, the prefix type constructors `(->)`, `[]`, `()`, `(,)`, and so on, always denote the built-in type constructors; they cannot be qualified, nor mentioned in import or export lists (Chapter 5). (Hence the special production, "gtycon", above.)

Although the list and tuple types have special syntax, their semantics is the same as the equivalent user-defined algebraic data types.

Notice that expressions and types have a consistent syntax. If t_i is the type of expression or pattern e_i, then the expressions `(\` e_1 `->` e_2`)`, `[`e_1`]`, and `(`e_1`,`e_2`)` have the types `(`t_1 `->` t_2`)`, `[`t_1`]`, and `(`t_1`,`t_2`)`, respectively.

With one exception (that of the distinguished type variable in a class declaration (Section 4.3.1)), the type variables in a Haskell type expression are all assumed to be universally quantified; there is no explicit syntax for universal quantification [5]. For example, the type expression `a -> a` denotes the type $\forall\, a.\ a \rightarrow a$. For clarity, however, we often write quantification explicitly when discussing the types of Haskell programs. When we write an explicitly quantified type, the scope of the $\forall$ extends as far to the right as possible; for example, $\forall\, a.\ a \rightarrow a$ means $\forall\, a.\ (a \rightarrow a)$.

4.1.3 Syntax of Class Assertions and Contexts

context	$\rightarrow$	*class*	
	$\mid$	`(` *class*$_1$ `,` ... `,` *class*$_n$ `)`	$(n \geq 0)$
class	$\rightarrow$	*qtycls tyvar*	
	$\mid$	*qtycls* `(` *tyvar atype*$_1$ ... *atype*$_n$ `)`	$(n \geq 1)$
qtycls	$\rightarrow$	[*modid* `.`] *tycls*	
tycls	$\rightarrow$	*conid*	
tyvar	$\rightarrow$	*varid*	

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

A *class assertion* has form $qtycls\ tyvar$, and indicates the membership of the type $tyvar$ in the class $qtycls$. A class identifier begins with an uppercase letter. A *context* consists of zero or more class assertions, and has the general form

$$(C_1\ u_1,\ \ldots,\ C_n\ u_n)$$

where $C_1,\ \ldots,\ C_n$ are class identifiers, and each of the $u_1,\ \ldots,\ u_n$ is either a type variable, or the application of type variable to one or more types. The outer parentheses may be omitted when $n = 1$. In general, we use cx to denote a context and we write cx => t to indicate the type t restricted by the context cx. The context cx must only contain type variables referenced in t. For convenience, we write cx => t even if the context cx is empty, although in this case the concrete syntax contains no =>.

4.1.4 Semantics of Types and Classes

In this subsection, we provide informal details of the type system. (Wadler and Blott [16] and Jones [10] discuss type and constructor classes, respectively, in more detail.)

The Haskell type system attributes a *type* to each expression in the program. In general, a type is of the form $\forall\,\overline{u}.\ cx\ \Rightarrow\ t$, where $\overline{u}$ is a set of type variables $u_1,\ \ldots,\ u_n$. In any such type, any of the universally-quantified type variables u_i that are free in cx must also be free in t. Furthermore, the context cx must be of the form given above in Section 4.1.3. For example, here are some valid types:

```
Eq a => a -> a
(Eq a, Show a, Eq b) => [a] -> [b] -> String
(Eq (f a), Functor f) => (a -> b) -> f a -> f b -> Bool
```

In the third type, the constraint `Eq (f a)` cannot be made simpler because `f` is universally quantified.

The type of an expression e depends on a *type environment* that gives types for the free variables in e, and a *class environment* that declares which types are instances of which classes (a type becomes an instance of a class only via the presence of an `instance` declaration or a `deriving` clause).

Types are related by a generalization preorder (specified below); the most general type, up to the equivalence induced by the generalization preorder, that can be assigned to a particular expression (in a given environment) is called its *principal type*. Haskell's extended Hindley–Milner type system can infer the principal type of all expressions, including the proper use of overloaded class methods (although certain ambiguous overloadings could arise, as described in Section 4.3.4). Therefore, explicit typings (called *type signatures*) are usually optional (see Sections 3.16 and 4.4.1).

The type $\forall\,\overline{u}.\ cx_1\ \Rightarrow\ t_1$ is *more general than* the type $\forall\,\overline{w}.\ cx_2\ \Rightarrow\ t_2$ if and only if there is a substitution S whose domain is $\overline{u}$ such that:

- t_2 is identical to $S(t_1)$.

- Whenever cx_2 holds in the class environment, $S(cx_1)$ also holds.

A value of type $\forall\,\overline{u}.\ cx\ \Rightarrow\ t$, may be instantiated at types $\overline{s}$ if and only if the context $cx[\overline{s}/\overline{u}]$ holds. For example, consider the function `double`:

```
double x = x + x
```

The most general type of `double` is $\forall\ a.$ `Num` $a\ \Rightarrow\ a\ \rightarrow\ a$. `double` may be applied to values of type `Int` (instantiating a to `Int`), since `Num Int` holds, because `Int` is an instance of the class `Num`. However, `double` may not normally be applied to values of type `Char`, because `Char` is not normally an instance of class `Num`. The user may choose to declare such an instance, in which case `double` may indeed be applied to a `Char`.

4.2 User-Defined Datatypes

In this section, we describe algebraic datatypes (`data` declarations), renamed datatypes (`newtype` declarations), and type synonyms (`type` declarations). These declarations may only appear at the top level of a module.

4.2.1 Algebraic Datatype Declarations

topdecl	→	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
simpletype	→	*tycon* $tyvar_1$ … $tyvar_k$	$(k \geq 0)$
constrs	→	$constr_1$ `|` … `|` $constr_n$	$(n \geq 1)$
constr	→	*con* [`!`] $atype_1$ … [`!`] $atype_k$	(arity $con\ =\ k,\ k \geq 0$)
	|	(*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*)	(infix *conop*)
	|	*con* `{` $fielddecl_1$ `,` … `,` $fielddecl_n$ `}`	$(n \geq 0)$
fielddecl	→	*vars* `::` (*type* | `!` *atype*)	
deriving	→	`deriving` (*dclass* | `(`$dclass_1$ `,` … `,` $dclass_n$`)`)	$(n \geq 0)$
dclass	→	*qtycls*	

The precedence for *constr* is the same as that for expressions – normal constructor application has higher precedence than infix constructor application (thus `a : Foo a` parses as `a : (Foo a)`).

An algebraic datatype declaration has the form:

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ K_1\ t_{11}\ \ldots\ t_{1k_1}\ |\ \cdots\ |\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$

where cx is a context. This declaration introduces a new *type constructor* T with one or more constituent *data constructors* $K_1,\ \ldots,\ K_n$. In this Report, the unqualified term "constructor" always means "data constructor".

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

The types of the data constructors are given by:

$$K_i :: \forall\ u_1 \ \ldots\ u_k.\ cx_i \Rightarrow t_{i1} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow (T\ u_1\ \ldots\ u_k)$$

where cx_i is the largest subset of cx that constrains only those type variables free in the types $t_{i1}, \ldots, t_{ik_i}$. The type variables u_1 through u_k must be distinct and may appear in cx and the t_{ij}; it is a static error for any other type variable to appear in cx or on the right-hand-side. The new type constant T has a kind of the form $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow *$ where the kinds κ_i of the argument variables u_i are determined by kind inference as described in Section 4.6. This means that T may be used in type expressions with anywhere between 0 and k arguments.

For example, the declaration

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

introduces a type constructor `Set` of kind $* \rightarrow *$, and constructors `NilSet` and `ConsSet` with types

$$\begin{array}{ll} \texttt{NilSet} & :: \forall\ a.\ \texttt{Set}\ a \\ \texttt{ConsSet} & :: \forall\ a.\ \texttt{Eq}\ a \Rightarrow a \rightarrow \texttt{Set}\ a \rightarrow \texttt{Set}\ a \end{array}$$

In the example given, the overloaded type for `ConsSet` ensures that `ConsSet` can only be applied to values whose type is an instance of the class `Eq`. Pattern matching against `ConsSet` also gives rise to an `Eq a` constraint. For example:

```
f (ConsSet a s) = a
```

the function `f` has inferred type `Eq a => Set a -> a`. The context in the `data` declaration has no other effect whatsoever.

The visibility of a datatype's constructors (i.e. the "abstractness" of the datatype) outside of the module in which the datatype is defined is controlled by the form of the datatype's name in the export list as described in Section 5.8.

The optional `deriving` part of a `data` declaration has to do with *derived instances*, and is described in Section 4.3.3.

Labelled Fields A data constructor of arity k creates an object with k components. These components are normally accessed positionally as arguments to the constructor in expressions or patterns. For large datatypes it is useful to assign *field labels* to the components of a data object. This allows a specific field to be referenced independently of its location within the constructor.

A constructor definition in a `data` declaration may assign labels to the fields of the constructor, using the record syntax (`C { ... }`). Constructors using field labels may be freely mixed with constructors without them. A constructor with associated field labels may still be used as an ordinary constructor; features using labels are simply a shorthand for operations using an underlying positional constructor. The arguments to the positional constructor occur in the same order as the labeled fields. For example, the declaration

```
data C = F { f1,f2 :: Int, f3 :: Bool }
```

defines a type and constructor identical to the one produced by

```
data C = F Int Int Bool
```

Operations using field labels are described in Section 3.15. A `data` declaration may use the same field label in multiple constructors as long as the typing of the field is the same in all cases after type synonym expansion. A label cannot be shared by more than one type in scope. Field names share the top level namespace with ordinary variables and class methods and must not conflict with other top level names in scope.

The pattern `F {}` matches any value built with constructor `F`, *whether or not* `F` *was declared with record syntax.*

Strictness Flags Whenever a data constructor is applied, each argument to the constructor is evaluated if and only if the corresponding type in the algebraic datatype declaration has a strictness flag, denoted by an exclamation point, "`!`". Lexically, "`!`" is an ordinary varsym not a $reservedop$; it has special significance only in the context of the argument types of a data declaration.

Translation: A declaration of the form

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ \ldots\ \texttt{|}\ K\ s_1\ \ldots\ s_n\ \texttt{|}\ \ldots$$

where each s_i is either of the form $\texttt{!}\,t_i$ or t_i, replaces every occurrence of K in an expression by

$$\texttt{(\textbackslash}\ x_1\ \ldots\ x_n\ \texttt{->}\ (\ ((K\ op_1\ x_1)\ op_2\ x_2)\ \ldots)\ op_n\ x_n)$$

where op_i is the non-strict apply function `$` if s_i is of the form t_i, and op_i is the strict apply function `$!` (see Section 6.2) if s_i is of the form $\texttt{!}\ t_i$. Pattern matching on K is not affected by strictness flags.

4.2.2 Type Synonym Declarations

$topdecl$	$\rightarrow$	$\texttt{type}\ simpletype\ \texttt{=}\ type$	
$simpletype$	$\rightarrow$	$tycon\ tyvar_1\ \ldots\ tyvar_k$	$(k \geq 0)$

A type synonym declaration introduces a new type that is equivalent to an old type. It has the form

$$\texttt{type}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ t$$

which introduces a new type constructor, T. The type $(T\ t_1\ \ldots t_k)$ is equivalent to the type $t[t_1/u_1,\ \ldots,\ t_k/u_k]$. The type variables u_1 through u_k must be distinct and are scoped only over t; it is a static error for any other type variable to appear in t. The kind of the new type constructor T

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

is of the form $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow \kappa$ where the kinds κ_i of the arguments u_i and κ of the right hand side t are determined by kind inference as described in Section 4.6. For example, the following definition can be used to provide an alternative way of writing the list type constructor:

```
type List = []
```

Type constructor symbols T introduced by type synonym declarations cannot be partially applied; it is a static error to use T without the full number of arguments.

Although recursive and mutually recursive datatypes are allowed, this is not so for type synonyms, *unless an algebraic datatype intervenes*. For example,

```
type Rec a   =  [Circ a]
data Circ a  =  Tag [Rec a]
```

is allowed, whereas

```
type Rec a   =  [Circ a]        -- invalid
type Circ a  =  [Rec a]         -- invalid
```

is not. Similarly, `type Rec a = [Rec a]` is not allowed.

Type synonyms are a convenient, but strictly syntactic, mechanism to make type signatures more readable. A synonym and its definition are completely interchangeable, except in the instance type of an `instance` declaration (Section 4.3.2).

4.2.3 Datatype Renamings

topdecl	$\rightarrow$	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
newconstr	$\rightarrow$	*con atype*	
	|	*con* `{` *var* `::` *type* `}`	
simpletype	$\rightarrow$	*tycon* $tyvar_1$ ... $tyvar_k$	$(k \geq 0)$

A declaration of the form

$$\texttt{newtype}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ N\ t$$

introduces a new type whose representation is the same as an existing type. The type `(`$T\ u_1 \ldots u_k$`)` renames the datatype t. It differs from a type synonym in that it creates a distinct type that must be explicitly coerced to or from the original type. Also, unlike type synonyms, `newtype` may be used to define recursive types. The constructor N in an expression coerces a value from type t to type `(`$T\ u_1\ \ldots\ u_k$`)`. Using N in a pattern coerces a value from type `(`$T\ u_1\ \ldots\ u_k$`)` to type t. These coercions may be implemented without execution time overhead; `newtype` does not change the underlying representation of an object.

New instances (see Section 4.3.2) can be defined for a type defined by `newtype` but may not be defined for a type synonym. A type created by `newtype` differs from an algebraic datatype in that the representation of an algebraic datatype has an extra level of indirection. This difference

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

may make access to the representation less efficient. The difference is reflected in different rules for pattern matching (see Section 3.17). Unlike algebraic datatypes, the newtype constructor N is *unlifted*, so that $N\ \perp$ is the same as $\perp$.

The following examples clarify the differences between `data` (algebraic datatypes), `type` (type synonyms), and `newtype` (renaming types.) Given the declarations

```
data D1 = D1 Int
data D2 = D2 !Int
type S = Int
newtype N = N Int
d1 (D1 i) = 42
d2 (D2 i) = 42
s i = 42
n (N i) = 42
```

the expressions (`d1` $\perp$), (`d2` $\perp$) and (`d2` (`D2` $\perp$)) are all equivalent to $\perp$, whereas (`n` $\perp$), (`n` (`N` $\perp$)), (`d1` (`D1` $\perp$)) and (`s` $\perp$) are all equivalent to `42`. In particular, (`N` $\perp$) is equivalent to $\perp$ while (`D1` $\perp$) is not equivalent to $\perp$.

The optional deriving part of a `newtype` declaration is treated in the same way as the deriving component of a `data` declaration; see Section 4.3.3.

A `newtype` declaration may use field-naming syntax, though of course there may only be one field. Thus:

```
newtype Age = Age { unAge :: Int }
```

brings into scope both a constructor and a de-constructor:

```
Age   :: Int -> Age
unAge :: Age -> Int
```

4.3 Type Classes and Overloading

4.3.1 Class Declarations

topdecl	$\rightarrow$	`class` [*scontext* `=>`] *tycls tyvar* [`where` *cdecls*]	
scontext	$\rightarrow$	*simpleclass*	
	$\vert$	`(` $simpleclass_1$ `,` ... `,` $simpleclass_n$ `)`	$(n \geq 0)$
simpleclass	$\rightarrow$	*qtycls tyvar*	
cdecls	$\rightarrow$	`{` $cdecl_1$ `;` ... `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	$\rightarrow$	*gendecl*	
	$\vert$	(*funlhs* $\vert$ *var*) *rhs*	

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

A *class declaration* introduces a new class and the operations (*class methods*) on it. A class declaration has the general form:

$$\texttt{class}\ cx\ \texttt{=>}\ C\ u\ \texttt{where}\ cdecls$$

This introduces a new class name C; the type variable u is scoped only over the class method signatures in the class body. The context cx specifies the superclasses of C, as described below; the only type variable that may be referred to in cx is u.

The superclass relation must not be cyclic, i.e. it must form a directed acyclic graph.

The $cdecls$ part of a `class` declaration contains three kinds of declarations:

- The class declaration introduces new *class methods* v_i, whose scope extends outside the `class` declaration. The class methods of a class declaration are precisely the v_i for which there is an explicit type signature

  $$v_i\ \texttt{::}\ cx_i\ \texttt{=>}\ t_i$$

  in $cdecls$. Class methods share the top level namespace with variable bindings and field names; they must not conflict with other top level bindings in scope. That is, a class method can not have the same name as a top level definition, a field name, or another class method.

  The type of the top-level class method v_i is:

  $$v_i :: \forall u, \overline{w}.\ (C u, cx_i) \Rightarrow t_i$$

  The t_i must mention u; it may mention type variables $\overline{w}$ other than u, in which case the type of v_i is polymorphic in both u and $\overline{w}$. The cx_i may constrain only $\overline{w}$; in particular, the cx_i may not constrain u. For example:

  ```
  class Foo a where
    op :: Num b => a -> b -> a
  ```

  Here the type of `op` is $\forall\ a,\ b.\ (\texttt{Foo}\ a,\ \texttt{Num}\ b)\ \Rightarrow\ a\ \rightarrow\ b\ \rightarrow\ a$.

- The $cdecls$ may also contain a *fixity declaration* for any of the class methods (but for no other values). However, since class methods declare top-level values, the fixity declaration for a class method may alternatively appear at top level, outside the class declaration.

- Lastly, the $cdecls$ may contain a *default class method* for any of the v_i. The default class method for v_i is used if no binding for it is given in a particular `instance` declaration (see Section 4.3.2). The default method declaration is a normal value definition, except that the left hand side may only be a variable or function definition. For example:

  ```
  class Foo a where
    op1, op2 :: a -> a
    (op1, op2) = ...
  ```

  is not permitted, because the left hand side of the default declaration is a pattern.

Other than these cases, no other declarations are permitted in $cdecls$.

A `class` declaration with no `where` part may be useful for combining a collection of classes into a larger one that inherits all of the class methods in the original ones. For example:

```
class  (Read a, Show a) => Textual a
```

In such a case, if a type is an instance of all superclasses, it is not *automatically* an instance of the subclass, even though the subclass has no immediate class methods. The `instance` declaration must be given explicitly with no `where` part.

4.3.2 Instance Declarations

$topdecl$	$\rightarrow$	`instance` [$scontext$ `=>`] $qtycls$ $inst$ [`where` $idecls$]	
$inst$	$\rightarrow$	$gtycon$	
	$\mid$	`(` $gtycon$ $tyvar_1$ ... $tyvar_k$ `)`	$(k \geq 0,\ tyvars$ distinct$)$
	$\mid$	`(` $tyvar_1$ `,` ... `,` $tyvar_k$ `)`	$(k \geq 2,\ tyvars$ distinct$)$
	$\mid$	`[` $tyvar$ `]`	
	$\mid$	`(` $tyvar_1$ `->` $tyvar_2$ `)`	$(tyvar_1$ and $tyvar_2$ distinct$)$
$idecls$	$\rightarrow$	`{` $idecl_1$ `;` ... `;` $idecl_n$ `}`	$(n \geq 0)$
$idecl$	$\rightarrow$	$(funlhs \mid var)\ rhs$	
	$\mid$		$(empty)$

An *instance declaration* introduces an instance of a class. Let

`class` cx `=>` $C\ u$ `where {` $cbody$ `}`

be a `class` declaration. The general form of the corresponding instance declaration is:

`instance` cx' `=>` $C\ (T\ u_1\ \ldots\ u_k)$ `where {` d `}`

where $k \geq 0$. The type $(T\ u_1\ \ldots\ u_k)$ must take the form of a type constructor T applied to simple type variables $u_1, \ldots u_k$; furthermore, T must not be a type synonym, and the u_i must all be distinct.

This prohibits instance declarations such as:

```
instance C (a,a) where ...
instance C (Int,a) where ...
instance C [[a]] where ...
```

The declarations d may contain bindings only for the class methods of C. It is illegal to give a binding for a class method that is not in scope, but the name under which it is in scope is immaterial; in particular, it may be a qualified name. (This rule is identical to that used for subordinate names in export lists – Section 5.2.) For example, this is legal, even though `range` is in scope only with the qualified name `Ix.range`.

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

```
module A where
  import qualified Ix

  instance Ix.Ix T where
    range = ...
```

The declarations may not contain any type signatures or fixity declarations, since these have already been given in the `class` declaration. As in the case of default class methods (Section 4.3.1), the method declarations must take the form of a variable or function definition.

If no binding is given for some class method then the corresponding default class method in the `class` declaration is used (if present); if such a default does not exist then the class method of this instance is bound to `undefined` and no compile-time error results.

An `instance` declaration that makes the type T to be an instance of class C is called a *C-T instance declaration* and is subject to these static restrictions:

- A type may not be declared as an instance of a particular class more than once in the program.
- The class and type must have the same kind; this can be determined using kind inference as described in Section 4.6.
- Assume that the type variables in the instance type $(T\ u_1\ \ldots\ u_k)$ satisfy the constraints in the instance context cx'. Under this assumption, the following two conditions must also be satisfied:

  1. The constraints expressed by the superclass context $cx[(T\ u1\ \ldots\ uk)/u]$ of C must be satisfied. In other words, T must be an instance of each of C's superclasses and the contexts of all superclass instances must be implied by cx'.
  2. Any constraints on the type variables in the instance type that are required for the class method declarations in d to be well-typed must also be satisfied.

  In fact, except in pathological cases it is possible to infer from the instance declaration the most general instance context cx' satisfying the above two constraints, but it is nevertheless mandatory to write an explicit instance context.

The following example illustrates the restrictions imposed by superclass instances:

```
class Foo a => Bar a where ...

instance (Eq a, Show a) => Foo [a] where ...

instance Num a => Bar [a] where ...
```

This example is valid Haskell. Since `Foo` is a superclass of `Bar`, the second instance declaration is only valid if `[a]` is an instance of `Foo` under the assumption `Num a`. The first instance declaration does indeed say that `[a]` is an instance of `Foo` under this assumption, because `Eq` and `Show` are superclasses of `Num`.

If the two instance declarations instead read like this:

```
instance Num a => Foo [a] where ...

instance (Eq a, Show a) => Bar [a] where ...
```

then the program would be invalid. The second instance declaration is valid only if `[a]` is an instance of `Foo` under the assumptions `(Eq a, Show a)`. But this does not hold, since `[a]` is only an instance of `Foo` under the stronger assumption `Num a`.

Further examples of `instance` declarations may be found in Chapter 8.

4.3.3 Derived Instances

As mentioned in Section 4.2.1, `data` and `newtype` declarations contain an optional `deriving` form. If the form is included, then *derived instance declarations* are automatically generated for the datatype in each of the named classes. These instances are subject to the same restrictions as user-defined instances. When deriving a class C for a type T, instances for all superclasses of C must exist for T, either via an explicit `instance` declaration or by including the superclass in the `deriving` clause.

Derived instances provide convenient commonly-used operations for user-defined datatypes. For example, derived instances for datatypes in the class `Eq` define the operations `==` and `/=`, freeing the programmer from the need to define them.

The only classes in the Prelude for which derived instances are allowed are `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`, all mentioned in Figure 6.1, page 85. The precise details of how the derived instances are generated for each of these classes are provided in Chapter 10, including a specification of when such derived instances are possible. Classes defined by the standard libraries may also be derivable.

A static error results if it is not possible to derive an `instance` declaration over a class named in a `deriving` form. For example, not all datatypes can properly support class methods in `Enum`. It is also a static error to give an explicit `instance` declaration for a class that is also derived.

If the `deriving` form is omitted from a `data` or `newtype` declaration, then *no* instance declarations are derived for that datatype; that is, omitting a `deriving` form is equivalent to including an empty deriving form: `deriving ()`.

4.3.4 Ambiguous Types, and Defaults for Overloaded Numeric Operations

$$\mathit{topdecl} \quad \rightarrow \quad \texttt{default (}\mathit{type}_1\texttt{ , } \ldots \texttt{ , }\mathit{type}_n\texttt{)} \qquad\qquad (n \geq 0)$$

A problem inherent with Haskell-style overloading is the possibility of an *ambiguous type*. For example, using the `read` and `show` functions defined in Chapter 10, and supposing that just `Int` and `Bool` are members of `Read` and `Show`, then the expression

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

```
let x = read "..." in show x  -- invalid
```

is ambiguous, because the types for `show` and `read`,

$$\begin{array}{lll}\texttt{show} & :: & \forall\, a.\, \texttt{Show}\ a\ \Rightarrow\ a\ \rightarrow\ \texttt{String}\\ \texttt{read} & :: & \forall\, a.\, \texttt{Read}\ a\ \Rightarrow\ \texttt{String}\ \rightarrow\ a\end{array}$$

could be satisfied by instantiating `a` as either `Int` in both cases, or `Bool`. Such expressions are considered ill-typed, a static error.

We say that an expression `e` has an *ambiguous type* if, in its type $\forall\, \overline{u}.\ cx\ \Rightarrow\ t$, there is a type variable u in $\overline{u}$ that occurs in cx but not in t. Such types are invalid.

For example, the earlier expression involving `show` and `read` has an ambiguous type since its type is $\forall\, a.\ \texttt{Show}\ a,\ \texttt{Read}\ a\ \Rightarrow\ \texttt{String}$.

Ambiguous types can only be circumvented by input from the user. One way is through the use of *expression type-signatures* as described in Section 3.16. For example, for the ambiguous expression given earlier, one could write:

```
let x = read "..." in show (x::Bool)
```

which disambiguates the type.

Occasionally, an otherwise ambiguous expression needs to be made the same type as some variable, rather than being given a fixed type with an expression type-signature. This is the purpose of the function `asTypeOf` (Chapter 8): x `` `asTypeOf` `` y has the value of x, but x and y are forced to have the same type. For example,

```
approxSqrt x = encodeFloat 1 (exponent x `div` 2) `asTypeOf` x
```

(See Section 6.4.6 for a description of `encodeFloat` and `exponent`.)

Ambiguities in the class `Num` are most common, so Haskell provides another way to resolve them – with a *default declaration*:

$$\texttt{default}\ (t_1\ ,\ \ldots\ ,\ t_n)$$

where $n \geq 0$, and each t_i must be a type for which $\texttt{Num}\ t_i$ holds. In situations where an ambiguous type is discovered, an ambiguous type variable, v, is defaultable if:

- v appears only in constraints of the form $C\ v$, where C is a class, and
- at least one of these classes is a numeric class, (that is, `Num` or a subclass of `Num`), and
- all of these classes are defined in the Prelude or a standard library (Figures 6.2 and 6.3, pages 93–94 show the numeric classes, and Figure 6.1, page 85, shows the classes defined in the Prelude.)

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

Each defaultable variable is replaced by the first type in the default list that is an instance of all the ambiguous variable's classes. It is a static error if no such type is found.

Only one default declaration is permitted per module, and its effect is limited to that module. If no default declaration is given in a module then it assumed to be:

```
default (Integer, Double)
```

The empty default declaration, `default ()`, turns off all defaults in a module.

4.4 Nested Declarations

The following declarations may be used in any declaration list, including the top level of a module.

4.4.1 Type Signatures

gendecl	→	*vars* `::` [*context* `=>`] *type*	
vars	→	*var*$_1$ `,` ... `,` *var*$_n$	$(n \geq 1)$

A type signature specifies types for variables, possibly with respect to a context. A type signature has the form:

$$v_1,\ \ldots,\ v_n\ \texttt{::}\ cx\ \texttt{=>}\ t$$

which is equivalent to asserting v_i `::` cx `=>` t for each i from 1 to n. Each v_i must have a value binding in the same declaration list that contains the type signature; i.e. it is invalid to give a type signature for a variable bound in an outer scope. Moreover, it is invalid to give more than one type signature for one variable, even if the signatures are identical.

As mentioned in Section 4.1.2, every type variable appearing in a signature is universally quantified over that signature, and hence the scope of a type variable is limited to the type signature that contains it. For example, in the following declarations

```
f :: a -> a
f x = x :: a                    -- invalid
```

the `a`'s in the two type signatures are quite distinct. Indeed, these declarations contain a static error, since `x` does not have type $\forall\ a.\ a$. (The type of `x` is dependent on the type of `f`; there is currently no way in Haskell to specify a signature for a variable with a dependent type; this is explained in Section 4.5.4.)

If a given program includes a signature for a variable f, then each use of f is treated as having the declared type. It is a static error if the same type cannot also be inferred for the defining occurrence of f.

If a variable f is defined without providing a corresponding type signature declaration, then each use of f outside its own declaration group (see Section 4.5) is treated as having the corresponding inferred, or *principal* type. However, to ensure that type inference is still possible, the defining occurrence, and all uses of f within its declaration group must have the same monomorphic type (from which the principal type is obtained by generalization, as described in Section 4.5.2).

For example, if we define

```
sqr x  =  x*x
```

then the principal type is `sqr` $::\ \forall\, a.$ `Num` $a \Rightarrow a \rightarrow a$, which allows applications such as `sqr 5` or `sqr 0.1`. It is also valid to declare a more specific type, such as

```
sqr :: Int -> Int
```

but now applications such as `sqr 0.1` are invalid. Type signatures such as

```
sqr :: (Num a, Num b) => a -> b     -- invalid
sqr :: a -> a                       -- invalid
```

are invalid, as they are more general than the principal type of `sqr`.

Type signatures can also be used to support *polymorphic recursion*. The following definition is pathological, but illustrates how a type signature can be used to specify a type more general than the one that would be inferred:

```
data T a  =  K (T Int) (T a)
f         :: T a -> a
f (K x y) =  if f x == 1 then f y else undefined
```

If we remove the signature declaration, the type of `f` will be inferred as `T Int -> Int` due to the first recursive call for which the argument to `f` is `T Int`. Polymorphic recursion allows the user to supply the more general type signature, `T a -> a`.

4.4.2 Fixity Declarations

$gendecl$	$\rightarrow$	$fixity$ $[integer]$ ops	
$fixity$	$\rightarrow$	`infixl` | `infixr` | `infix`	
ops	$\rightarrow$	op_1 `,` $\ldots$ `,` op_n	$(n \geq 1)$
op	$\rightarrow$	$varop$ | $conop$	

A fixity declaration gives the fixity and binding precedence of one or more operators. The $integer$ in a fixity declaration must be in the range 0 to 9. A fixity declaration may appear anywhere that a type signature appears and, like a type signature, declares a property of a particular operator. Also like a type signature, a fixity declaration can only occur in the same sequence of declarations as the declaration of the operator itself, and at most one fixity declaration may be given for any

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

Table 4.1: Precedences and fixities of prelude operators

Precedence	Left associative operators	Non-associative operators	Right associative operators
9	`!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `‘div‘`, `‘mod‘`, `‘rem‘`, `‘quot‘`		
6	`+`, `-`		
5			`:`, `++`
4		`==`, `/=`, `<`, `<=`, `>`, `>=`, `‘elem‘`, `‘notElem‘`	
3			`&&`
2			`||`
1	`>>`, `>>=`		
0			`$`, `$!`, `‘seq‘`

operator. (Class methods are a minor exception; their fixity declarations can occur either in the class declaration itself or at top level.)

There are three kinds of fixity, non-, left- and right-associativity (`infix`, `infixl`, and `infixr`, respectively), and ten precedence levels, 0 to 9 inclusive (level 0 binds least tightly, and level 9 binds most tightly). If the *digit* is omitted, level 9 is assumed. Any operator lacking a fixity declaration is assumed to be `infixl 9` (See Section 3 for more on the use of fixities). Table 4.1 lists the fixities and precedences of the operators defined in the Prelude.

Fixity is a property of a particular entity (constructor or variable), just like its type; fixity is not a property of that entity's *name*. For example:

```
module Bar( op ) where
  infixr 7 ‘op‘
  op = ...

module Foo where
  import qualified Bar
  infix 3 ‘op‘

  a ‘op‘ b = (a ‘Bar.op‘ b) + 1

  f x = let
           p ‘op‘ q = (p ‘Foo.op‘ q) * 2
        in ...
```

Here, `‘Bar.op‘` is `infixr 7`, `‘Foo.op‘` is `infix 3`, and the nested definition of `op` in `f`'s right-hand side has the default fixity of `infixl 9`. (It would also be possible to give a fixity to the nested definition of `‘op‘` with a nested fixity declaration.)

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

4.4.3 Function and Pattern Bindings

$$
\begin{array}{lcl}
decl & \rightarrow & (funlhs \mid pat^{0})\ rhs \\
\\
funlhs & \rightarrow & var\ apat\ \{\ apat\ \} \\
 & \mid & pat^{i+1}\ varop^{(a,i)}\ pat^{i+1} \\
 & \mid & lpat^{i}\ varop^{(\mathrm{l},i)}\ pat^{i+1} \\
 & \mid & pat^{i+1}\ varop^{(\mathrm{r},i)}\ rpat^{i} \\
 & \mid & \texttt{(}\ funlhs\ \texttt{)}\ apat\ \{\ apat\ \} \\
\\
rhs & \rightarrow & \texttt{=}\ exp\ [\texttt{where}\ decls] \\
 & \mid & gdrhs\ [\texttt{where}\ decls] \\
\\
gdrhs & \rightarrow & gd\ \texttt{=}\ exp\ [gdrhs] \\
\\
gd & \rightarrow & \texttt{|}\ exp^{0}
\end{array}
$$

We distinguish two cases within this syntax: a *pattern binding* occurs when the left hand side is a pat^{0}; otherwise, the binding is called a *function binding*. Either binding may appear at the top-level of a module or within a `where` or `let` construct.

4.4.3.1 Function bindings.

A function binding binds a variable to a function value. The general form of a function binding for variable x is:

$$
\begin{array}{lll}
x & p_{11}\ \ldots\ p_{1k} & match_{1} \\
\ldots & & \\
x & p_{n1}\ \ldots\ p_{nk} & match_{n}
\end{array}
$$

where each p_{ij} is a pattern, and where each $match_i$ is of the general form:

$$\texttt{=}\ e_i\ \texttt{where}\ \texttt{\{}\ decls_i\ \texttt{\}}$$

or

$$
\begin{array}{ll}
\texttt{|}\ g_{i1} & \texttt{=}\ e_{i1} \\
\ldots & \\
\texttt{|}\ g_{im_i} & \texttt{=}\ e_{im_i} \\
 & \texttt{where}\ \texttt{\{}\ decls_i\ \texttt{\}}
\end{array}
$$

and where $n \geq 1$, $1 \leq i \leq n$, $m_i \geq 1$. The former is treated as shorthand for a particular case of the latter, namely:

$$\texttt{|}\ \texttt{True}\ \texttt{=}\ e_i\ \texttt{where}\ \texttt{\{}\ decls_i\ \texttt{\}}$$

Note that all clauses defining a function must be contiguous, and the number of patterns in each clause must be the same. The set of patterns corresponding to each match must be *linear* – no variable is allowed to appear more than once in the entire set.

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

Alternative syntax is provided for binding functional values to infix operators. For example, these three function definitions are all equivalent:

```
plus x y z = x+y+z
x `plus` y = \ z -> x+y+z
(x `plus` y) z = x+y+z
```

Translation: The general binding form for functions is semantically equivalent to the equation (i.e. simple pattern binding):

$$
\begin{array}{l}
x\ \texttt{=}\ \backslash\ x_1\ \ldots\ x_k\ \texttt{->}\ \texttt{case}\ \texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_k\texttt{)}\ \texttt{of}\ \texttt{(}p_{11}\texttt{,}\ \ldots\texttt{,}\ p_{1k}\texttt{)}\ match_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{(}p_{n1}\texttt{,}\ \ldots\texttt{,}\ p_{nk}\texttt{)}\ match_n
\end{array}
$$

where the x_i are new identifiers.

4.4.3.2 Pattern bindings.

A pattern binding binds variables to values. A *simple* pattern binding has form $p = e$. The pattern p is matched "lazily" as an irrefutable pattern, as if there were an implicit ~ in front of it. See the translation in Section 3.12.

The *general* form of a pattern binding is $p\ match$, where a $match$ is the same structure as for function bindings above; in other words, a pattern binding is:

$$
\begin{array}{lll}
p & \mid g_1 & = e_1 \\
 & \mid g_2 & = e_2 \\
 & \ldots & \\
 & \mid g_m & = e_m \\
 & \multicolumn{2}{l}{\texttt{where}\ \{\ decls\ \}}
\end{array}
$$

Translation: The pattern binding above is semantically equivalent to this simple pattern binding:

$$
\begin{array}{lll}
p & \texttt{=} & \texttt{let}\ decls\ \texttt{in} \\
 & & \texttt{if}\ g_1\ \texttt{then}\ e_1\ \texttt{else} \\
 & & \texttt{if}\ g_2\ \texttt{then}\ e_2\ \texttt{else} \\
 & & \ldots \\
 & & \texttt{if}\ g_m\ \texttt{then}\ e_m\ \texttt{else error "Unmatched pattern"}
\end{array}
$$

A note about syntax. It is usually straightforward to tell whether a binding is a pattern binding or a function binding, but the existence of `n+k` patterns sometimes confuses the issue. Here are four examples:

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

```
x + 1 = ...             -- Function binding, defines (+)
                        -- Equivalent to   (+) x 1 = ...

(x + 1) = ...           -- Pattern binding, defines x

(x + 1) * y = ...       -- Function binding, defines (*)
                        -- Equivalent to   (*) (x+1) y = ...

(x + 1) y = ...         -- Function binding, defines (+)
                        -- Equivalent to   (+) x 1 y = ...
```

The first two can be distinguished because a pattern binding has a pat^0 on the left hand side, not a pat – the former cannot be an unparenthesised `n+k` pattern.

4.5 Static Semantics of Function and Pattern Bindings

The static semantics of the function and pattern bindings of a `let` expression or `where` clause are discussed in this section.

4.5.1 Dependency Analysis

In general the static semantics are given by the normal Hindley–Milner inference rules. A *dependency analysis transformation* is first performed to increase polymorphism. Two variables bound by value declarations are in the same *declaration group* if either

1. they are bound by the same pattern binding, or
2. their bindings are mutually recursive (perhaps via some other declarations that are also part of the group).

Application of the following rules causes each `let` or `where` construct (including the `where` defining the top level bindings in a module) to bind only the variables of a single declaration group, thus capturing the required dependency analysis:[1]

1. The order of declarations in `where`/`let` constructs is irrelevant.
2. `let {`d_1`;` d_2`} in` e = `let {`d_1`} in (let {`d_2`} in` e`)`
   (when no identifier bound in d_2 appears free in d_1)

[1]A similar transformation is described in Peyton Jones' book [14].

4.5.2 Generalization

The Hindley–Milner type system assigns types to a `let`-expression in two stages. First, the right-hand side of the declaration is typed, giving a type with no universal quantification. Second, all type variables that occur in this type are universally quantified unless they are associated with bound variables in the type environment; this is called *generalization*. Finally, the body of the `let`-expression is typed.

For example, consider the declaration

```
f x = let g y = (y,y)
      in ...
```

The type of `g`'s definition is $a \rightarrow (a, a)$. The generalization step attributes to `g` the polymorphic type $\forall\ a.\ a \rightarrow (a, a)$, after which the typing of the "`...`" part can proceed.

When typing overloaded definitions, all the overloading constraints from a single declaration group are collected together, to form the context for the type of each variable declared in the group. For example, in the definition:

```
f x = let g1 x y = if x>y then show x else g2 y x
          g2 p q = g1 q p
      in ...
```

The types of the definitions of `g1` and `g2` are both $a \rightarrow a \rightarrow$ `String`, and the accumulated constraints are `Ord` a (arising from the use of `>`), and `Show` a (arising from the use of `show`). The type variables appearing in this collection of constraints are called the *constrained type variables*.

The generalization step attributes to both `g1` and `g2` the type

$$\forall\ a. (\texttt{Ord}\ a,\ \texttt{Show}\ a) \Rightarrow a \rightarrow a \rightarrow \texttt{String}$$

Notice that `g2` is overloaded in the same way as `g1` even though the occurrences of `>` and `show` are in the definition of `g1`.

If the programmer supplies explicit type signatures for more than one variable in a declaration group, the contexts of these signatures must be identical up to renaming of the type variables.

4.5.3 Context Reduction Errors

As mentioned in Section 4.1.4, the context of a type may constrain only a type variable, or the application of a type variable to one or more types. Hence, types produced by generalization must be expressed in a form in which all context constraints have be reduced to this "head normal form". Consider, for example, the definition:

```
f xs y  =  xs == [y]
```

Its type is given by

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

and not

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

Even though the equality is taken at the list type, the context must be simplified, using the instance declaration for `Eq` on lists, before generalization. If no such instance is in scope, a static error occurs.

Here is an example that shows the need for a constraint of the form $C\ (m\ t)$ where m is one of the type variables being generalized; that is, where the class C applies to a type expression that is not a type variable or a type constructor. Consider:

```
f :: (Monad m, Eq (m a)) => a -> m a -> Bool
f x y = return x == y
```

The type of `return` is `Monad m => a -> m a`; the type of `(==)` is `Eq a => a -> a -> Bool`. The type of `f` should be therefore `(Monad m, Eq (m a)) => a -> m a -> Bool`, and the context cannot be simplified further.

The instance declaration derived from a data type `deriving` clause (see Section 4.3.3) must, like any instance declaration, have a *simple* context; that is, all the constraints must be of the form $C\ a$, where a is a type variable. For example, in the type

```
data Apply a b = App (a b)  deriving Show
```

the derived Show instance will produce a context `Show (a b)`, which cannot be reduced and is not simple; thus a static error results.

4.5.4 Monomorphism

Sometimes it is not possible to generalize over all the type variables used in the type of the definition. For example, consider the declaration

```
f x = let g y z = ([x,y], z)
      in ...
```

In an environment where `x` has type a, the type of `g`'s definition is $a \rightarrow b \rightarrow ([a], b)$. The generalization step attributes to `g` the type $\forall\ b.\ a \rightarrow b \rightarrow ([a], b)$; only b can be universally quantified because a occurs in the type environment. We say that the type of `g` is *monomorphic in the type variable a*.

The effect of such monomorphism is that the first argument of all applications of `g` must be of a single type. For example, it would be valid for the "`...`" to be

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

```
(g True, g False)
```

(which would, incidentally, force `x` to have type `Bool`) but invalid for it to be

```
(g True, g 'c')
```

In general, a type $\forall\,\overline{u}.\ cx\ \Rightarrow\ t$ is said to be *monomorphic* in the type variable a if a is free in $\forall\,\overline{u}.\ cx\ \Rightarrow\ t$.

It is worth noting that the explicit type signatures provided by Haskell are not powerful enough to express types that include monomorphic type variables. For example, we cannot write

```
f x = let
        g :: a -> b -> ([a],b)
        g y z = ([x,y], z)
      in ...
```

because that would claim that `g` was polymorphic in both `a` and `b` (Section 4.4.1). In this program, `g` can only be given a type signature if its first argument is restricted to a type not involving type variables; for example

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

This signature would also cause `x` to have type `Int`.

4.5.5 The Monomorphism Restriction

Haskell places certain extra restrictions on the generalization step, beyond the standard Hindley–Milner restriction described above, which further reduces polymorphism in particular cases.

The monomorphism restriction depends on the binding syntax of a variable. Recall that a variable is bound by either a *function binding* or a *pattern binding*, and that a *simple* pattern binding is a pattern binding in which the pattern consists of only a single variable (Section 4.4.3).

The following two rules define the monomorphism restriction:

The monomorphism restriction

Rule 1. We say that a given declaration group is *unrestricted* if and only if:

(a): every variable in the group is bound by a function binding or a simple pattern binding (Section 4.4.3.2), *and*

(b): an explicit type signature is given for every variable in the group that is bound by simple pattern binding.

The usual Hindley–Milner restriction on polymorphism is that only type variables that do not occur free in the environment may be generalized. In addition, *the constrained type variables of a restricted declaration group may not be generalized* in the generalization step for that group. (Recall that a type variable is constrained if it must belong to some type class; see Section 4.5.2.)

Rule 2. Any monomorphic type variables that remain when type inference for an entire module is complete, are considered *ambiguous*, and are resolved to particular types using the defaulting rules (Section 4.3.4).

Motivation Rule 1 is required for two reasons, both of which are fairly subtle.

- *Rule 1 prevents computations from being unexpectedly repeated:* e.g. `genericLength` is a standard function (in library `List`) whose type is given by

  ```
  genericLength :: Num a => [b] -> a
  ```

  Now consider the following expression:

  ```
  let { len = genericLength xs } in (len, len)
  ```

  It looks as if `len` should be computed only once, but without Rule 1 it might be computed twice, once at each of two different overloadings. If the programmer does actually wish the computation to be repeated, an explicit type signature may be added:

  ```
  let { len :: Num a => a; len = genericLength xs } in (len, len)
  ```

- *Rule 1 prevents ambiguity:* e.g. consider the declaration group

  ```
  [(n,s)] = reads t
  ```

  Recall that `reads` is a standard function whose type is given by the signature

  ```
  reads :: (Read a) => String -> [(a,String)]
  ```

  Without Rule 1, `n` would be assigned the type $\forall\ a.$ `Read` $a\ \Rightarrow\ a$ and `s` the type $\forall\ a.$ `Read` $a \Rightarrow$ `String`. The latter is an invalid type, because it is inherently ambiguous. It is not possible to determine at what overloading to use `s`, nor can this be solved by adding a type signature for `s`. Hence, when *non-simple* pattern bindings are used (Section 4.4.3.2), the types inferred are always monomorphic in their constrained type variables, irrespective of whether a type signature is provided. In this case, both `n` and `s` are monomorphic in a.

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

The same constraint applies to pattern-bound functions. For example, in

```
(f,g) = ((+),(-))
```

both `f` and `g` are monomorphic regardless of any type signatures supplied for `f` or `g`.

Rule 2 is required because there is no way to enforce monomorphic use of an *exported* binding, except by performing type inference on modules outside the current module. Rule 2 states that the exact types of all the variables bound in a module must be determined by that module alone, and not by any modules that import it.

```
module M1(len1) where
  default( Int, Double )
  len1 = genericLength "Hello"

module M2 where
  import M1(len1)
  len2 = (2*len1) :: Rational
```

When type inference on module `M1` is complete, `len1` has the monomorphic type `Num a => a` (by Rule 1). Rule 2 now states that the monomorphic type variable `a` is ambiguous, and must be resolved using the defaulting rules of Section 4.3.4. Hence, `len1` gets type `Int`, and its use in `len2` is type-incorrect. (If the above code is actually what is wanted, a type signature on `len1` would solve the problem.)

This issue does not arise for nested bindings, because their entire scope is visible to the compiler.

Consequences The monomorphism rule has a number of consequences for the programmer. Anything defined with function syntax usually generalizes as a function is expected to. Thus in

```
f x y = x+y
```

the function `f` may be used at any overloading in class `Num`. There is no danger of recomputation here. However, the same function defined with pattern syntax:

```
f = \x -> \y -> x+y
```

requires a type signature if `f` is to be fully overloaded. Many functions are most naturally defined using simple pattern bindings; the user must be careful to affix these with type signatures to retain full overloading. The standard prelude contains many examples of this:

```
sum  :: (Num a) => [a] -> a
sum  =  foldl (+) 0
```

Rule 1 applies to both top-level and nested definitions. Consider

```
module M where
  len1 = genericLength "Hello"
  len2 = (2*len1) :: Rational
```

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press

Here, type inference finds that `len1` has the monomorphic type (`Num a => a`); and the type variable `a` is resolved to `Rational` when performing type inference on `len2`.

4.6 Kind Inference

This section describes the rules that are used to perform *kind inference*, i.e. to calculate a suitable kind for each type constructor and class appearing in a given program.

The first step in the kind inference process is to arrange the set of datatype, synonym, and class definitions into dependency groups. This can be achieved in much the same way as the dependency analysis for value declarations that was described in Section 4.5. For example, the following program fragment includes the definition of a datatype constructor `D`, a synonym `S` and a class `C`, all of which would be included in the same dependency group:

```
data C a => D a = Foo (S a)
type S a = [D a]
class C a where
    bar :: a -> D a -> Bool
```

The kinds of variables, constructors, and classes within each group are determined using standard techniques of type inference and kind-preserving unification [10]. For example, in the definitions above, the parameter `a` appears as an argument of the function constructor `(->)` in the type of `bar` and hence must have kind $*$. It follows that both `D` and `S` must have kind $* \rightarrow *$ and that every instance of class `C` must have kind $*$.

It is possible that some parts of an inferred kind may not be fully determined by the corresponding definitions; in such cases, a default of $*$ is assumed. For example, we could assume an arbitrary kind κ for the `a` parameter in each of the following examples:

```
data App f a = A (f a)
data Tree a  = Leaf | Fork (Tree a) (Tree a)
```

This would give kinds $(\kappa \rightarrow *) \rightarrow \kappa \rightarrow *$ and $\kappa \rightarrow *$ for `App` and `Tree`, respectively, for any kind κ, and would require an extension to allow polymorphic kinds. Instead, using the default binding $\kappa = *$, the actual kinds for these two constructors are $(* \rightarrow *) \rightarrow * \rightarrow *$ and $* \rightarrow *$, respectively.

Defaults are applied to each dependency group without consideration of the ways in which particular type constructor constants or classes are used in later dependency groups or elsewhere in the program. For example, adding the following definition to those above does not influence the kind inferred for `Tree` (by changing it to $(* \rightarrow *) \rightarrow *$, for instance), and instead generates a static error because the kind of `[]`, $* \rightarrow *$, does not match the kind $*$ that is expected for an argument of `Tree`:

```
type FunnyTree = Tree []     -- invalid
```

This is important because it ensures that each constructor and class are used consistently with the same kind whenever they are in scope.

https://doi.org/10.1017/S0956796803000613 Published online by Cambridge University Press