JFP **13** (1): 67–80, January 2003.
DOI: 10.1017/S0956796803000716 Printed in the United Kingdom

Chapter 5

Modules

A module defines a collection of values, datatypes, type synonyms, classes, etc. (see Chapter 4), in an environment created by a set of *imports* (resources brought into scope from other modules). It *exports* some of these resources, making them available to other modules. We use the term *entity* to refer to a value, type, or class defined in, imported into, or perhaps exported from a module.

A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, which must be a computation of type `IO` τ for some type τ (see Chapter 7). When the program is executed, the computation `main` is performed, and its result (of type τ) is discarded.

Modules may reference other modules via explicit `import` declarations, each giving the name of a module to be imported and specifying its entities to be imported. Modules may be mutually recursive.

Modules are used for name-space control, and are not first class values. A multi-module Haskell program can be converted into a single-module program by giving each entity a unique name, changing all occurrences to refer to the appropriate unique name, and then concatenating all the module bodies[1]. For example, here is a three-module program:

[1]There are two minor exceptions to this statement. First, `default` declarations scope over a single module (see Section 4.3.4). Second, Rule 2 of the monomorphism restriction (see Section 4.5.5) is affected by module boundaries.

```
module Main where
  import A
  import B
  main = A.f >> B.f

module A where
  f = ...

module B where
  f = ...
```

It is equivalent to the following single-module program:

```
module Main where
  main = af >> bf

  af = ...

  bf = ...
```

Because they are allowed to be mutually recursive, modules allow a program to be partitioned freely without regard to dependencies.

The name-space for modules themselves is flat, with each module being associated with a unique module name (which are Haskell identifiers beginning with a capital letter; i.e. *modid*). There is one distinguished module, `Prelude`, which is imported into all modules by default (see Section 5.6), plus a set of standard library modules that may be imported as required (see Part II).

5.1 Module Structure

A module defines a mutually recursive scope containing declarations for value bindings, data types, type synonyms, classes, etc. (see Chapter 4).

module	→	`module` *modid* [*exports*] `where` *body*	
	|	*body*	
body	→	`{` *impdecls* `;` *topdecls* `}`	
	|	`{` *impdecls* `}`	
	|	`{` *topdecls* `}`	
modid	→	*conid*	
impdecls	→	$impdecl_1$ `;` ... `;` $impdecl_n$	$(n \geq 1)$
topdecls	→	$topdecl_1$ `;` ... `;` $topdecl_n$	$(n \geq 1)$

A module begins with a header: the keyword `module`, the module name, and a list of entities (enclosed in round parentheses) to be exported. The header is followed by a possibly-empty list of `import` declarations (*impdecls*, Section 5.3) that specify modules to be imported, optionally

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

restricting the imported bindings. This is followed by a possibly-empty list of top-level declarations ($topdecls$, Chapter 4).

An abbreviated form of module, consisting only of the module body, is permitted. If this is used, the header is assumed to be '`module Main(main) where`'. If the first lexeme in the abbreviated module is not a `{`, then the layout rule applies for the top level of the module.

5.2 Export Lists

$$
\begin{array}{lcll}
exports & \rightarrow & \texttt{(}\ export_1\ \texttt{,}\ \ldots\ \texttt{,}\ export_n\ [\ \texttt{,}\ ]\ \texttt{)} & (n \geq 0) \\
\\
export & \rightarrow & qvar & \\
 & | & qtycon\ [\texttt{(..)} \mid \texttt{(}\ cname_1\ \texttt{,}\ \ldots\ \texttt{,}\ cname_n\ \texttt{)}] & (n \geq 0) \\
 & | & qtycls\ [\texttt{(..)} \mid \texttt{(}\ var_1\ \texttt{,}\ \ldots\ \texttt{,}\ var_n\ \texttt{)}] & (n \geq 0) \\
 & | & \texttt{module}\ modid & \\
\\
cname & \rightarrow & var \mid con &
\end{array}
$$

An *export list* identifies the entities to be exported by a module declaration. A module implementation may only export an entity that it declares, or that it imports from some other module. If the export list is omitted, all values, types and classes defined in the module are exported, *but not those that are imported*.

Entities in an export list may be named as follows:

1. A value, field name, or class method, whether declared in the module body or imported, may be named by giving the name of the value as a $qvarid$, which must be in scope. Operators should be enclosed in parentheses to turn them into $qvarid$s.

2. An algebraic datatype T declared by a `data` or `newtype` declaration may be named in one of three ways:

   - The form T names the type *but not the constructors or field names*. The ability to export a type without its constructors allows the construction of abstract datatypes (see Section 5.8).
   - The form T`(`c_1`,` $\ldots$`,`c_n`)`, names the type and some or all of its constructors and field names.
   - The abbreviated form T`(..)` names the type and all its constructors and field names that are currently in scope (whether qualified or not).

   In all cases, the (possibly-qualified) type constructor T must be in scope. The constructor and field names c_i in the second form are unqualified; one of these subordinate names is legal if and only if (a) it names a constructor or field of T, and (b) the constructor or field is in

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

scope in the module body *regardless of whether it is in scope under a qualified or unqualified name*. For example, the following is legal

```
module A( Mb.Maybe( Nothing, Just ) ) where
  import qualified Maybe as Mb
```

Data constructors cannot be named in export lists except as subordinate names, because they cannot otherwise be distinguished from type constructors.

3. A type synonym T declared by a `type` declaration may be named by the form T, where T is in scope.

4. A class C with operations $f_1, \ldots, f_n$ declared in a `class` declaration may be named in one of three ways:

   - The form C names the class *but not the class methods*.
   - The form C`(`f_1`,` $\ldots$`,`f_n`)`, names the class and some or all of its methods.
   - The abbreviated form C`(..)` names the class and all its methods that are in scope (whether qualified or not).

   In all cases, C must be in scope. In the second form, one of the (unqualified) subordinate names f_i is legal if and only if (a) it names a class method of C, and (b) the class method is in scope in the module body regardless of whether it is in scope under a qualified or unqualified name.

5. The form "`module M`" names the set of all entities that are in scope with both an unqualified name *e* and a qualified name `M`.*e*. This set may be empty. For example:

   ```
   module Queue( module Stack, enqueue, dequeue ) where
       import Stack
       ...
   ```

   Here the module `Queue` uses the module name `Stack` in its export list to abbreviate all the entities imported from `Stack`.

   A module can name its own local definitions in its export list using its own name in the "`module M`" syntax, because a local declaration brings into scope both a qualified and unqualified name (Section 5.5.1). For example:

   ```
   module Mod1( module Mod1, module Mod2 ) where
   import Mod2
   import Mod3
   ```

   Here module `Mod1` exports all local definitions as well as those imported from `Mod2` but not those imported from `Mod3`.

   It is an error to use `module M` in an export list unless `M` is the module bearing the export list, or `M` is imported by at least one import declaration (qualified or unqualified).

Exports lists are cumulative: the set of entities exported by an export list is the union of the entities exported by the individual items of the list.

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

It makes no difference to an importing module how an entity was exported. For example, a field name `f` from data type `T` may be exported individually (`f`, item (1) above); or as an explicitly-named member of its data type (`T(f)`, item (2)); or as an implicitly-named member (`T(..)`, item(2)); or by exporting an entire module (`module M`, item (5)).

The *unqualified* names of the entities exported by a module must all be distinct (within their respective namespace). For example

```
module A ( C.f, C.g, g, module B ) where   -- an invalid module
import B(f)
import qualified C(f,g)
g = f True
```

There are no name clashes within module `A` itself, but there are name clashes in the export list between `C.g` and `g` (assuming `C.g` and `g` are different entities – remember, modules can import each other recursively), and between `module B` and `C.f` (assuming `B.f` and `C.f` are different entities).

5.3 Import Declarations

$$
\begin{array}{lcll}
impdecl & \rightarrow & \texttt{import}\ [\texttt{qualified}]\ modid\ [\texttt{as}\ modid]\ [impspec] & \\
 & | & & (empty\ declaration) \\
impspec & \rightarrow & \texttt{(}\ import_1\ \texttt{,}\ \ldots\ \texttt{,}\ import_n\ [\ \texttt{,}\ ]\ \texttt{)} & (n \geq 0) \\
 & | & \texttt{hiding (}\ import_1\ \texttt{,}\ \ldots\ \texttt{,}\ import_n\ [\ \texttt{,}\ ]\ \texttt{)} & (n \geq 0) \\
 & & & \\
import & \rightarrow & var & \\
 & | & tycon\ [\ \texttt{(..)}\ |\ \texttt{(}\ cname_1\ \texttt{,}\ \ldots\ \texttt{,}\ cname_n\ \texttt{)}] & (n \geq 0) \\
 & | & tycls\ [\texttt{(..)}\ |\ \texttt{(}\ var_1\ \texttt{,}\ \ldots\ \texttt{,}\ var_n\ \texttt{)}] & (n \geq 0) \\
cname & \rightarrow & var\ |\ con &
\end{array}
$$

The entities exported by a module may be brought into scope in another module with an `import` declaration at the beginning of the module. The `import` declaration names the module to be imported and optionally specifies the entities to be imported. A single module may be imported by more than one `import` declaration. Imported names serve as top level declarations: they scope over the entire body of the module but may be shadowed by local non-top-level bindings.

The effect of multiple `import` declarations is strictly cumulative: an entity is in scope if it is imported by any of the `import` declarations in a module. The ordering of import declarations is irrelevant.

Lexically, the terminal symbols "`as`", "`qualified`" and "`hiding`" are each a *varid* rather than a *reservedid*. They have special significance only in the context of an `import` declaration; they may also be used as variables.

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

5.3.1 What is Imported

Exactly which entities are to be imported can be specified in one of the following three ways:

1. The imported entities can be specified explicitly by listing them in parentheses. Items in the list have the same form as those in export lists, except qualifiers are not permitted and the '`module` *modid*' entity is not permitted. When the `(..)` form of import is used for a type or class, the `(..)` refers to all of the constructors, methods, or field names exported from the module.

   The list must name only entities exported by the imported module. The list may be empty, in which case nothing except the instances is imported.

2. Entities can be excluded by using the form `hiding(` $import_1$ `,` ... `,` $import_n$ `)`, which specifies that all entities exported by the named module should be imported except for those named in the list. Data constructors may be named directly in hiding lists without being prefixed by the associated type. Thus, in

   ```
   import M hiding (C)
   ```

   any constructor, class, or type named `C` is excluded. In contrast, using `C` in an import list names only a class or type.

   It is an error to hide an entity that is not, in fact, exported by the imported module.

3. Finally, if *impspec* is omitted then all the entities exported by the specified module are imported.

5.3.2 Qualified Import

For each entity imported under the rules of Section 5.3.1, the top-level environment is extended. If the import declaration used the `qualified` keyword, only the *qualified name* of the entity is brought into scope. If the `qualified` keyword is omitted, then *both* the qualified *and* unqualified name of the entity is brought into scope. Section 5.5.1 describes qualified names in more detail.

The qualifier on the imported name is either the name of the imported module, or the local alias given in the `as` clause (Section 5.3.3) on the `import` statement. Hence, *the qualifier is not necessarily the name of the module in which the entity was originally declared.*

The ability to exclude the unqualified names allows full programmer control of the unqualified namespace: a locally defined entity can share the same name as a qualified import:

```
module Ring where
import qualified Prelude   -- All Prelude names must be qualified
import List( nub )

l1 + l2 = l1 Prelude.++ l2 -- + differs from that in the Prelude
l1 * l2 = nub (l1 + l2)    -- * differs from that in the Prelude

succ = (Prelude.+ 1)
```

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

5.3.3 Local Aliases

Imported modules may be assigned a local alias in the importing module using the `as` clause. For example, in

```
import qualified VeryLongModuleName as C
```

entities must be referenced using '`C.`' as a qualifier instead of '`VeryLongModuleName.`'. This also allows a different module to be substituted for `VeryLongModuleName` without changing the qualifiers used for the imported module. It is legal for more than one module in scope to use the same qualifier, provided that all names can still be resolved unambiguously. For example:

```
module M where
  import qualified Foo as A
  import qualified Baz as A
  x = A.f
```

This module is legal provided only that `Foo` and `Baz` do not both export `f`.

An `as` clause may also be used on an un-`qualified` `import` statement:

```
import Foo as A(f)
```

This declaration brings into scope `f` and `A.f`.

5.3.4 Examples

To clarify the above import rules, suppose the module `A` exports `x` and `y`. Then this table shows what names are brought into scope by the specified import statement:

Import declaration	Names brought into scope
`import A`	`x`, `y`, `A.x`, `A.y`
`import A()`	(nothing)
`import A(x)`	`x`, `A.x`
`import qualified A`	`A.x`, `A.y`
`import qualified A()`	(nothing)
`import qualified A(x)`	`A.x`
`import A hiding ()`	`x`, `y`, `A.x`, `A.y`
`import A hiding (x)`	`y`, `A.y`
`import qualified A hiding ()`	`A.x`, `A.y`
`import qualified A hiding (x)`	`A.y`
`import A as B`	`x`, `y`, `B.x`, `B.y`
`import A as B(x)`	`x`, `B.x`
`import qualified A as B`	`B.x`, `B.y`

In all cases, all instance declarations in scope in module `A` are imported (Section 5.4).

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

5.4 Importing and Exporting Instance Declarations

Instance declarations cannot be explicitly named on import or export lists. All instances in scope within a module are *always* exported and any import brings *all* instances in from the imported module. Thus, an instance declaration is in scope if and only if a chain of `import` declarations leads to the module containing the instance declaration.

For example, `import M()` does not bring any new names in scope from module `M`, but does bring in any instances visible in `M`. A module whose only purpose is to provide instance declarations can have an empty export list. For example

```
module MyInstances() where
  instance Show (a -> b) where
      show fn = "<<function>>"
  instance Show (IO a) where
      show io = "<<IO action>>"
```

5.5 Name Clashes and Closure

5.5.1 Qualified Names

A *qualified name* is written as $modid$ `.` $name$ (Section 2.4). A qualified name is brought into scope:

- *By a top level declaration.* A top-level declaration brings into scope both the unqualified *and* the qualified name of the entity being defined. Thus:

  ```
  module M where
    f x = ...
    g x = M.f x x
  ```

  is legal. The *defining* occurrence must mention the *unqualified* name; therefore, it is illegal to write

  ```
  module M where
    M.f x = ...                  -- ILLEGAL
    g x = let M.y = x+1 in ...   -- ILLEGAL
  ```

- *By an* `import` *declaration.* An `import` declaration, whether `qualified` or not, always brings into scope the qualified name of the imported entity (Section 5.3). This allows a qualified import to be replaced with an unqualified one without forcing changes in the references to the imported names.

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

5.5.2 Name Clashes

If a module contains a bound occurrence of a name, such as `f` or `A.f`, it must be possible unambiguously to resolve which entity is thereby referred to; that is, there must be only one binding for `f` or `A.f` respectively.

It is *not* an error for there to exist names that cannot be so resolved, provided that the program does not mention those names. For example:

```
module A where
  import B
  import C
  tup = (b, c, d, x)

module B( d, b, x, y ) where
  import D
  x = ...
  y = ...
  b = ...

module C( d, c, x, y ) where
  import D
  x = ...
  y = ...
  c = ...

module D( d ) where
  d = ...
```

Consider the definition of `tup`.

- The references to `b` and `c` can be unambiguously resolved to `b` declared in `B`, and `c` declared in `C` respectively.
- The reference to `d` is unambiguously resolved to `d` declared in `D`. In this case the same entity is brought into scope by two routes (the import of `B` and the import of `C`), and can be referred to in `A` by the names `d`, `B.d`, and `C.d`.
- The reference to `x` is ambiguous: it could mean `x` declared in `B`, or `x` declared in `C`. The ambiguity could be fixed by replacing the reference to `x` by `B.x` or `C.x`.
- There is no reference to `y`, so it is not erroneous that distinct entities called `y` are exported by both `B` and `C`. An error is only reported if `y` is actually mentioned.

The name occurring in a type signature or fixity declarations is always unqualified, and unambiguously refers to another declaration in the same declaration list (except that the fixity declaration for a class method can occur at top level – Section 4.4.2). For example, the following module is legal:

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

```
module F where

  sin :: Float -> Float
  sin x = (x::Float)

  f x = Prelude.sin (F.sin x)
```

The local declaration for `sin` is legal, even though the Prelude function `sin` is implicitly in scope. The references to `Prelude.sin` and `F.sin` must both be qualified to make it unambiguous which `sin` is meant. However, the unqualified name `sin` in the type signature in the first line of `F` unambiguously refers to the local declaration for `sin`.

5.5.3 Closure

Every module in a Haskell program must be *closed*. That is, every name explicitly mentioned by the source code must be either defined locally or imported from another module. However, entities that the compiler requires for type checking or other compile time analysis need not be imported if they are not mentioned by name. The Haskell compilation system is responsible for finding any information needed for compilation without the help of the programmer. That is, the import of a variable `x` does not require that the datatypes and classes in the signature of `x` be brought into the module along with `x` unless these entities are referenced by name in the user program. The Haskell system silently imports any information that must accompany an entity for type checking or any other purposes. Such entities need not even be explicitly exported: the following program is valid even though `T` does not escape `M1`:

```
module M1(x) where
  data T = T
  x = T

module M2 where
  import M1(x)
  y = x
```

In this example, there is no way to supply an explicit type signature for `y` since `T` is not in scope. Whether or not `T` is explicitly exported, module `M2` knows enough about `T` to correctly type check the program.

The type of an exported entity is unaffected by non-exported type synonyms. For example, in

```
module M(x) where
  type T = Int
  x :: T
  x = 1
```

the type of `x` is both `T` and `Int`; these are interchangeable even when `T` is not in scope. That is, the definition of `T` is available to any module that encounters it whether or not the name `T` is in scope. The only reason to export `T` is to allow other modules to refer it by name; the type checker finds the definition of `T` if needed whether or not it is exported.

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

5.6 Standard Prelude

Many of the features of Haskell are defined in Haskell itself as a library of standard datatypes, classes, and functions, called the "Standard Prelude." In Haskell, the Prelude is contained in the module `Prelude`. There are also many predefined library modules, which provide less frequently used functions and types. For example, complex numbers, arrays and most of the input/output are all part of the standard libraries. These are defined in Part II. Separating libraries from the Prelude has the advantage of reducing the size and complexity of the Prelude, allowing it to be more easily assimilated, and increasing the space of useful names available to the programmer.

Prelude and library modules differ from other modules in that their semantics (but not their implementation) are a fixed part of the Haskell language definition. This means, for example, that a compiler may optimize calls to functions in the Prelude without consulting the source code of the Prelude.

5.6.1 The `Prelude` Module

The `Prelude` module is imported automatically into all modules as if by the statement '`import Prelude`', if and only if it is not imported with an explicit `import` declaration. This provision for explicit import allows entities defined in the Prelude to be selectively imported, just like those from any other module.

The semantics of the entities in `Prelude` is specified by a reference implementation of `Prelude` written in Haskell, given in Chapter 8. Some datatypes (such as `Int`) and functions (such as `Int` addition) cannot be specified directly in Haskell. Since the treatment of such entities depends on the implementation, they are not formally defined in the appendix. The implementation of `Prelude` is also incomplete in its treatment of tuples: there should be an infinite family of tuples and their instance declarations, but the implementation only gives a scheme.

Chapter 8 defines the module `Prelude` using several other modules: `PreludeList`, `PreludeIO`, and so on. These modules are *not* part of Haskell 98, and they cannot be imported separately. They are simply there to help explain the structure of the `Prelude` module; they should be considered part of its implementation, not part of the language definition.

5.6.2 Shadowing Prelude Names

The rules about the Prelude have been cast so that it is possible to use Prelude names for nonstandard purposes; however, every module that does so must have an `import` declaration that makes this nonstandard usage explicit. For example:

```
module A( null, nonNull ) where
  import Prelude hiding( null )
  null, nonNull :: Int -> Bool
  null    x = x == 0
  nonNull x = not (null x)
```

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

Module `A` redefines `null`, and contains an unqualified reference to `null` on the right hand side of `nonNull`. The latter would be ambiguous without the `hiding(null)` on the `import Prelude` statement. Every module that imports `A` unqualified, and then makes an unqualified reference to `null` must also resolve the ambiguous use of `null` just as `A` does. Thus there is little danger of accidentally shadowing Prelude names.

It is possible to construct and use a different module to serve in place of the Prelude. Other than the fact that it is implicitly imported, the Prelude is an ordinary Haskell module; it is special only in that some objects in the Prelude are referenced by special syntactic constructs. Redefining names used by the Prelude does not affect the meaning of these special constructs. For example, in

```
module B where
  import Prelude()
  import MyPrelude
  f x = (x,x)
  g x = (,) x x
  h x = [x] ++ []
```

the explicit `import Prelude()` declaration prevents the automatic import of `Prelude`, while the declaration `import MyPrelude` brings the non-standard prelude into scope. The special syntax for tuples (such as `(x,x)` and `(,)`) and lists (such as `[x]` and `[]`) continues to refer to the tuples and lists defined by the standard `Prelude`; there is no way to redefine the meaning of `[x]`, for example, in terms of a different implementation of lists. On the other hand, the use of `++` is not special syntax, so it refers to `++` imported from `MyPrelude`.

It is not possible, however, to hide `instance` declarations in the `Prelude`. For example, one cannot define a new instance for `Show Char`.

5.7 Separate Compilation

Depending on the Haskell implementation used, separate compilation of mutually recursive modules may require that imported modules contain additional information so that they may be referenced before they are compiled. Explicit type signatures for all exported values may be necessary to deal with mutual recursion. The precise details of separate compilation are not defined by this report.

5.8 Abstract Datatypes

The ability to export a datatype without its constructors allows the construction of abstract datatypes (ADTs). For example, an ADT for stacks could be defined as:

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press

```
module Stack( StkType, push, pop, empty ) where
  data StkType a = EmptyStk | Stk a (StkType a)
  push x s = Stk x s
  pop (Stk _ s) = s
  empty = EmptyStk
```

Modules importing `Stack` cannot construct values of type `StkType` because they do not have access to the constructors of the type. Instead, they must use `push`, `pop`, and `empty` to construct such values.

It is also possible to build an ADT on top of an existing type by using a `newtype` declaration. For example, stacks can be defined with lists:

```
module Stack( StkType, push, pop, empty ) where
  newtype StkType a = Stk [a]
  push x (Stk s) = Stk (x:s)
  pop (Stk (_:s)) = Stk s
  empty = Stk []
```

https://doi.org/10.1017/S0956796803000716 Published online by Cambridge University Press