Modules and Abstract Data Types*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Presented by Ralf Sasse

Grouping definitions

- Until now, all our functions, types, and classes are in one file.
  - Does not scale to large programs (name space pollution).
  - To reuse a function in another project, you have to copy it.
- A **module** groups functions, types, and classes into a component.
  - Reuse by importing

```
import Data.List
import Data.Char
```

  - Mechanisms to control name space pollution.
  - Gains in abstraction by controlling exports.

    Hide internals by explicitly listing exports.

Running example: FIFO Queues

```
data Queue a = Queue [a]
  deriving (Show)

empty :: Queue a
empty = Queue []

isEmpty :: Queue a -> Bool
isEmpty (Queue xs) = null xs

push :: a -> Queue a -> Queue a
push x (Queue xs) = Queue (xs ++ [x])

pull :: Queue a -> (a, Queue a)
pull (Queue (x:xs)) = (x, Queue xs)
pull _              = error errEmptyQueue

top :: Queue a -> a
top (Queue (x:xs)) = x
top _              = error errEmptyQueue

errEmptyQueue = "Queue is empty"
```

```
? pull (push 3
        (push 2
        (push 1 empty)))
(1,Queue [2,3])
```

Modules

- Declaration: File starts with
  `module <module name> where`

  Convention: The module name is the same as the file name.

- Example:

```
module Queue where

data Queue a = Queue [a]

empty :: Queue a
empty = ...
```

- If there's no module declaration, Haskell automatically inserts `module Main where`.

  That's why `ghci`'s prompt says `*Main>`.

Export List

- The **export list** specifies which functions and types a client of the module can use.

```
module Queue
  ( Queue(..)   -- export Queue data type and all its constructors
  , empty, isEmpty, push, pull, top
  )             -- Do not export errQueueEmpty
where

data Queue a = Queue
...
```

- Corresponds to `public` modifier in Java, C++, and C#.

- Without export list, everything is exported.
  Every module should specify its exports explicitly.

Module import

- Import modules immediately after module declaration.

```
module Tree where

import qualified Queue as Q
import Data.Char (ord)

f x = ... ord ... Q.empty ...
g (Q.Queue xs) = ...
```

- Recommended forms of import:
  - **Qualified import:** `import qualified Queue as Q`
    import all exported names with explicit prefix
  - **Selective import:** `import Queue (push, pop)`
    specify list of imported names

- Prelude is imported automatically.

Abstract data types

- Constructors determine the representation of the data type.

  If exported, clients can pattern-match and directly access representation. They are like `public` fields in Java/C++/C#.

```
import qualified Queue as Q
g (Q.Queue xs) = ...              -- direct access to implementation
```

- Violates encapsulation:
  Cannot change implementation. Cannot ensure invariants.
- **Do not export datatype constructors** to get an ADT.
  - Provide functions to build values.
  - Provide accessor functions to destruct datatype (get methods).

```
module Queue (Queue,     -- export only datatype Queue, not constructors
  empty, isEmpty, push, pull, top, toList) where
toList (Queue xs) = xs
```

More Efficient Queues

```
module Queue (Queue, empty, isEmpty, push, pull, top, toList) where

-- second list caches pushed elements in reversed order until they are pulled
data Queue a = Queue [a] [a] deriving (Show)

toList (Queue xs ys) = xs ++ reverse ys
empty = Queue [] []
isEmpty (Queue xs ys) = null xs && null ys

push x (Queue xs ys) = Queue xs (x:ys)

pull (Queue []     []) = error "Queue is empty."
pull (Queue (x:xs) ys) = (x, Queue xs ys)
pull (Queue []     ys) = pull (Queue (reverse ys) [])

top = fst . pull
```

ADT ensures that clients work unchanged with both implementations.

```
? push 4 (push 3 (snd (pull (push 2 (push 1 empty)))))
Queue [2] [4,3]
```

Module Summary

```
module Queue (Queue, empty, push, toList) where
import Data.List (reverse)
data Queue a = Queue [a] [a]
empty = Queue [] []
toList (Queue xs ys) = xs ++ reverse ys
```

- Group types, functions, and classes into components.
- Export lists control exports.
- Qualified or selective imports avoid namespace pollution.
- **Encapsulation:** Hide datatype representation by not exporting constructors.
  - Export **functions**, not constructors, to build and destruct values.
  - Raise level of abstraction.