universität innsbruck

WS 2019/2020

Functional Programming

Part 6 – Modules and Abstract Datatypes

René Thiemann Benedikt Dornauer Maximilian W. Haslbeck
Bart Keulen Florian Meßner Jonas Schöpf
Vincent van Oostrom Xiang Zhang

Department of Computer Science

Overview

- scope of names
- modules
- abstract datatypes
- example: queues

Scope

- consider program (1 compile error)

```
radius = 15
area radius = pi^2 * radius

squares x = [ x^2 | x <- [0 .. x]]

length [] = []
length (_:xs) = 1 + length xs

data Rat = Rat Integer Integer
create_rat = normalize . Rat where normalize ... = ...
external_usage = Rat 2 4
```

- scope
  - resolve ambiguities
  - defines which names of variables, functions, types, ... are visible at a given program position
  - controlling scope to structure larger programs (imports / exports)

Scope of Names

```
radius = 15
area radius = pi^2 * radius
```

- in the following we assume that `name_i` in the real code is always just `name` and the `_i` is used for addressing the different occurrences of `name`
- renamed Haskell program
  ```
  radius_1 = 15
  area_1 radius_2 = pi_1^2 * radius_3
  ```
- scope of names in right-hand sides of equations
  - is `radius_3` referring to `radius_2` or `radius_1`?
  - to what is `pi_1` referring to?
- rule of thumb for searching `name`: search inside-out
  - think of abstract syntax tree of expression
  - whenever you pass a **`let`**, **`where`**, **`case`**, or function definition where `name` is bound, then refer to that local name
  - if nothing is found, then search global function `name`, also in Prelude
- `radius_3` refers to `radius_2`, `pi_1` to `Prelude.pi`

Local Names in Case-Expressions

- general case:

```
case exp of {pat_1 -> exp_1; ...; pat_n -> exp_n}
```

  - each `pat_i` binds the variables that occur in `pat_i`
  - these variables can be used in `exp_i`
  - the newly bound variables of `pat_i` bind stronger than any previously bound variables

- example Haskell expression

```
case xs_1 of                    -- renamed Haskell expr.
  [] -> xs_2
  (x_1 : xs_3) -> case xs_4 ++ ys_1 of
     [] -> ys_2
     (x_2 : xs_5) -> x_3 : xs_6 ++ ys_3
```

  - `x_3` refers to `x_2` (since `x_2` is further inside than `x_1`)
  - `xs_6` refers to `xs_5` (since `xs_5` is further inside than `xs_3`)
  - `xs_4` refers to `xs_3`
  - `xs_1`, `xs_2`, `ys_1`, `ys_2`, and `ys_3` are not bound in this expression (the proper references need to be determined further outside)

Local Names in Let-Expressions

- general case:

```
let {
  pat_1 = exp_1; ...; pat_n = exp_n;
  fn_1 pats_1 = fexp_1; ...; fn_m pats_m = fexp_m
} in exp
```

  - all variables in `pat_1` ... `pat_n` and all names `fn_1` ... `fn_m` are bound
  - these can be used in `exp`, in each `exp_i` and in each `fexp_j`
  - variables of `pats_j` bind strongest, but only in `fexp_j`

-
```
let (x_1, y_1) = (y_2 + 1, 5)  -- renamed Haskell expr.
    f_1 x_2 = x_3 + g_1 y_3 id_1
    g_2 y_4 f_2 = f_3 $ g_3 x_4 f_4
in (f_5, g_4, x_5, y_5)
```

  - `y_2`, `y_3` and `y_5` refer to `y_1`
  - `x_3` refers to `x_2` since `x_2` binds stronger than `x_1`
  - `x_4` and `x_5` refer to `x_1`
  - `f_3` and `f_4` refer to `f_2` since `f_2` binds stronger than `f_1`
  - `g_1`, `g_3` and `g_4` refer to `g_2`
  - `f_5` refers to `f_1`
  - `id_1` is not bound in this expression

Global Function Definitions

- general case:
  `fname pats = exp`
  - all variables in `pats` are bound locally and can be used in `exp`
  - `fname` is not locally bound, but added to global lookup table
  - all variables/names in `exp` without local reference will be looked up in global lookup table
  - lookup in global table does not permit ambiguities
-
```
radius_1 = 15        -- renamed Haskell prog.
area_2 radius_2 = pi_1^2 * radius_3
length_1 [] = []
length_2 (_:xs_1) = 1 + length_3 xs_2
```
  - `radius_1`, `area_2` and `length_1/2` are stored in global lookup table
  - global lookup table has ambiguity: `length_1/2` vs. `Prelude.length`
  - `pi_1` is not locally bound and therefore refers to `Prelude.pi`
  - `radius_3` refers to local `radius_2` and not to global `radius_1`
  - `xs_2` refers to `xs_1`
  - `length_3` is not locally bound and because of mentioned ambiguity, this leads to an compile error

Global vs Local Definitions

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

-- definition 1
length = foldr (\ _-> (1 +)) 0

-- definition 2
length =
  let { length [] = 0; length (x : xs) = 1 + length xs }
  in length

-- definition 3
length [] = 0
length (_ : xs) = 1 + length xs
```

- definitions 1 and 2 compile since there is no `length` on the rhs that needs a global lookup
- in contrast, definition 3 does not compile
- still definitions 1 and 2 result in ambiguities in global lookup table $\rightarrow$ study Haskell's module system

Modules

- so far
  - Haskell program is a single file, consisting of several definitions
  - all global definitions are visible to user

```
-- functions on rational numbers
data Rat = RatC Integer Integer
normalize (RatC n d) = ...              -- internal func.
createRat n d = normalize $ RatC n d -- external func.
...
-- one applications of rational numbers
-- approximate pi to a certain precision
pi_approx :: Integer -> Rat
pi_approx p = ...
```

- motivation for modules
  - structure programs into smaller reusable parts without copying
  - distinguish between internal and external definitions
    - clear interface for users of modules
    - maintain invariants
    - improve maintainability

Modules in Haskell

```
-- first line of file Module_Name.hs
module Module_Name(export_list) where
-- standard Haskell type and function definitions
```

- each `Module_Name` has to start with uppercase letter
- each module is usually stored in separate file `Module_Name.hs`
- if Haskell file contains no **`module`** declaration, ghci inserts module name `Main`
- `export_list` is comma-separated list of function-names and type-names,
  these functions and types will be accessible for users of the module
- if (`export_list`) is omitted, then everything is exported
- for types there are different export possibilities
  - **`module`** `Name(Type)` exports `Type`, but no constructors of `Type`
  - **`module`** `Name(Type(..))` exports `Type` and its constructors

Example: Rational Numbers

```
module Rat(Rat, create_Rat, numerator, denominator) where
data Rat = RatC Integer Integer
normalize = ...
create_Rat n d = normalize $ Rat n d
numerator (Rat n d) = n
...
instance Num Rat where ...
instance Show Rat where ...
```

- external users know that a type `Rat` exists
- they only see functions `create_Rat`, `numerator` and `denominator`
- they don't have access to `RatC` and therefore cannot form expressions like `RatC 2 4` which break invariant of cancelled fractions
- they can perform calculations with rational numbers since they have access to (+) of class `Num`, etc., in particular for the instance `Rat`
- for the same reason, they can display rational numbers via `show`

Example: Application

```
module Pi_Approx(pi_approx, Rat) where
-- Prelude is implicitly imported

-- import everything that is exported by module Rat
import Rat

-- or only import certain parts
import Rat(Rat, create_Rat)

-- import declarations must be before other definitions
pi_approx :: Integer -> Rat
pi_approx n = let init_approx = create_Rat 314 100 in ...
```

- there can be multiple `import` declarations
- what is imported is not automatically exported
  - when importing `Pi_Approx`, type `Rat` is visible, but `create_Rat` is not
  - if application requires both `Rat` and `Pi_Approx`, import both modules:
    ```
    import Pi_Approx
    import Rat
    ```

Resolving Ambiguities

```
-- Foo.hs
module Foo where pi = 3.1415

-- Problem.hs
module Problem where

import Foo

pi = 3.1415
area r = pi * r^2
```

- problem: what is `pi` in definition of `area`? (global name)
- lookup map is ambiguous: `pi` defined in `Prelude`, `Foo`, and `Problem`
- ambiguity persists, even if definition is identical
- solution via qualifier: disambiguate by using `Module_Name.name` instead of `name`
  - write `area r = Problem.pi * r^2` in `Problem.hs` (or `area r = Prelude.pi * r^2`)

Qualified Imports

```
module Foo where pi = 3.1415
module Some_Long_Module_Name where fun x = x + x

module Example_Qualified_Imports where

-- all imports of Foo have to use qualifier
import qualified Foo
-- result: no ambiguity on unqualified "pi"

import qualified Some_Long_Module_Name as S
-- "as"-syntax changes name of qualifier

area r = pi * r^2
myfun x = S.fun (x * x)
```

- further information on modules
  https://www.haskell.org/onlinereport/modules.html

Concrete and Abstract Datatypes

- concrete datatypes
  - defined via `data` which defines values of that type
  - user defines own operations on this type via pattern matching
  - no need for primitive operations on that type
  - examples: lists, `Rat`, `Temperature`, `Bool`, ...
- abstract datatypes
  - defined via their primitive operations
  - usually no access to internal structure of representation of values
  - pattern matching only via equality
  - abstraction barrier: internal structure can be easily changed
  - meaning of operations usually specified
  - examples: `Char`, `Integer`, `Double`, ... which provide basic arithmetic operations and conversion to strings

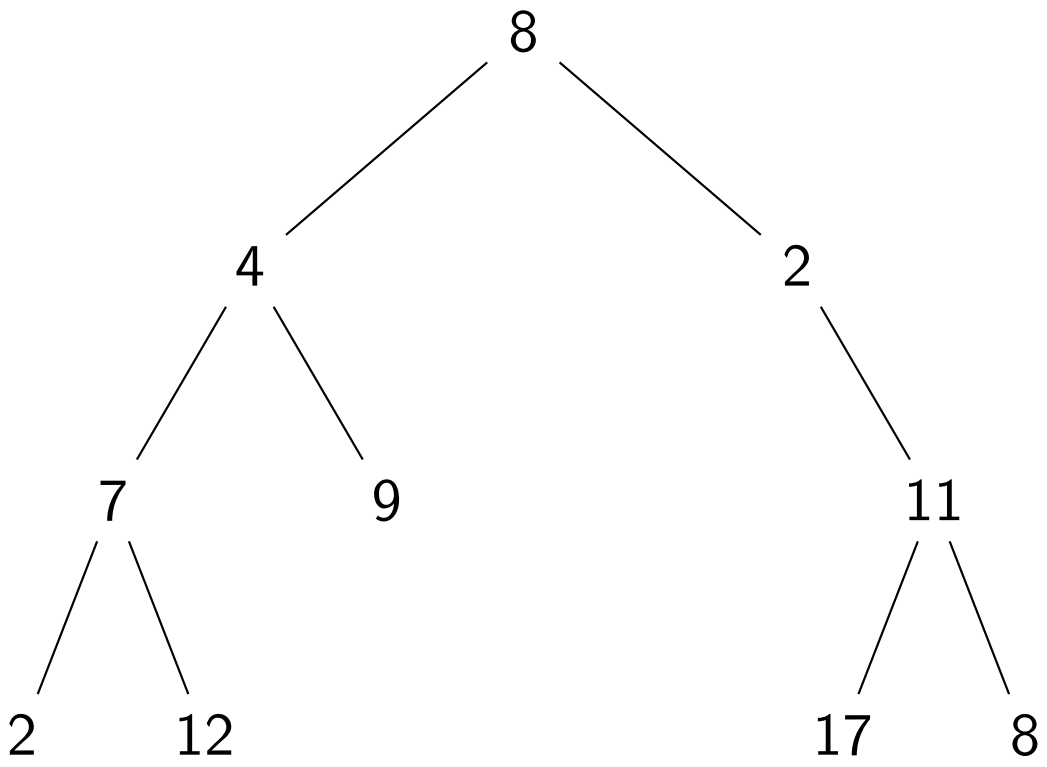
Example Abstract Datatype: Queues

- queues are useful in cs: printer (jobs), web-server (requests), ...
- queue provides the following operations
  - `empty :: Queue a` – the empty queue for elements of type `a`
  - `isEmpty :: Queue a -> Bool` – check whether queue is empty
  - `dequeue :: Queue a -> (a, Queue a)` – remove head of queue
  - `enqueue :: a -> Queue a -> Queue a` – add new element to end of queue

  these operations in combination with their types are the signature of the abstract datatype `Queue a`
- signature only gives idea about operations; more information can be specified via axiomatic specification in the form of equations or formulas
  - `isEmpty empty`
  - `not $ isEmpty $ enqueue x q`
  - `dequeue (enqueue x empty) = (x, empty)`
  - `not $ isEmpty q` $\longrightarrow$ `dequeue q = (y, q')` $\longrightarrow$ `dequeue (enqueue x q) = (y, enqueue x q')`

Example Application for Queues: Tree-Traversals

- consider tree



- tree-traversal: visit all nodes, e.g., to search for node, or convert nodes to list
  - in-order [2,7,12,4,9,8,2,17,11,8]
  - depth-first search, pre-order [8,4,7,2,12,9,2,11,17,8]
  - breadth-first search [8,4,2,7,9,11,2,12,17,8]

Tree Traversals in Haskell

```
data Tree a = Empty | Node (Tree a) a (Tree a)

inorder :: Tree a -> [a]
inorder Empty = []
inorder (Node l n r) = inorder l ++ [n] ++ inorder r

preorder :: Tree a -> [a]
preorder Empty = []
preorder (Node l n r) = [n] ++ preorder l ++ preorder r

bfs :: Tree a -> [a]
bfs t = bfsMain (enqueue t empty) where
  bfsMain :: Queue (Tree a) -> [a]
  bfsMain q
    | isEmpty q = []
    | otherwise = let (t', q') = dequeue q in
        case t' of
          Empty -> bfsMain q'
          Node l n r ->
            n : (bfsMain $ enqueue r . enqueue l $ q')
```

Implementing an Abstract Datatype

- implementation has to provide the desired operations and must satisfy the specification (informal text or axiomatic)
  - `empty :: Queue a`
  - `isEmpty :: Queue a -> Bool`
  - `dequeue :: Queue a -> (a, Queue a)`
  - `enqueue :: a -> Queue a -> Queue a`
  - `isEmpty empty`
  - `not $ isEmpty $ enqueue x q`
  - `dequeue (enqueue x empty) = (x, empty)`
  - `not $ isEmpty q` $\longrightarrow$ `dequeue q = (y, q')` $\longrightarrow$ `dequeue (enqueue x q) = (y, enqueue x q')`
- any implementation can be used, e.g., basic one in the beginning, which might be replaced by more efficient one later on
- if corner cases are not specified, implementation can choose freely, e.g., how dequeue should behave on empty queues
- modules can be used to hide internals

A Basic Implementation of Queues

```
module Basic_Queue(Queue, empty, isEmpty, dequeue, enqueue)
  where

data Queue a = Empty | Enqueue a (Queue a)

empty = Empty
enqueue = Enqueue

isEmpty Empty = True
isEmpty (Enqueue x q) = False

dequeue (Enqueue x Empty) = (x, Empty)
dequeue (Enqueue x q) = (y, Enqueue x q') where
  (y, q') = dequeue q
dequeue Empty = error "dequeue on empty queue"
```

- implementation is rather direct translation of specification
- `empty` and `enqueue` are implemented as constructors of queues, and exported; still the constructors itself are not exported and so internal structure is not revealed, e.g., externally no pattern matching possible

Notes on the Basic Implementation of Queues

```
...
data Queue a = Empty | Enqueue a (Queue a)

isEmpty Empty = True
isEmpty (Enqueue x q) = False

dequeue (Enqueue x Empty) = (x, Empty)
dequeue (Enqueue x q) = (y, Enqueue x q') where
  (y, q') = dequeue q
dequeue Empty = error "dequeue on empty queue"
```

- we did not prove that implementation meets the axiomatic specification; will be covered in
  - program verification (bsc), or
  - interactive theorem proving (msc)
- implementation is inefficient, since first enqueuing n elements and then dequeueing n elements requires $\sim \frac{1}{2}n^2$ evaluation steps

Towards a More Efficient Implementation of Queues

- previous queue-type is essentially a list where the list head represents the end of the queue (queue = reversed list)
- assume customers 1, 2, 3 and 4 enqueue in that order, then the representation is `[4, 3, 2, 1]`
- enqueuing is efficient since it just adds element in front of list
- dequeuing is expensive since it traverses and rebuilds whole list
- new version: store queue as pair of two lists: (front, rear)
  - front part of queue (head of queue is head of list)
  - rear part of queue in reverse order (tail of queue is head of list)
  - invariant: front part of queue is empty implies whole queue is empty
- example queue with customers 1, 2, 3, 4 has multiple representations
  - `([1,2,3,4], [])` ✔
  - `([1,2,3], [4])` ✔
  - `([1,2], [4,3])` ✔
  - `([1], [4,3,2])` ✔
  - `([], [4,3,2,1])` ✘
- advantage: often constant time access to both ends of queue

More Efficient Implementation of Queues

```haskell
module Better_Queue(Queue, empty, isEmpty, dequeue, enqueue)
  where

type Queue a = ([a], [a])

empty :: Queue a
empty = ([], [])

isEmpty :: Queue a -> Bool
isEmpty (front, _) = null front

enqueue :: a -> Queue a -> Queue a
enqueue x (front, rear) = maybe_mtf (front, x : rear)

dequeue :: Queue a -> (a, Queue a)
dequeue ([], _) = error "dequeue on empty queue"
dequeue (x : front, rear) = (x, maybe_mtf (front, rear))

maybe_mtf ([], rear) = (reverse rear, [])
maybe_mtf q = q
```

Efficiency of More Efficient Implementation

```
dequeue ([], _) = error "dequeue on empty queue"
dequeue (x : front, rear) = (x, maybe_mtf (front, rear))

maybe_mtf ([], rear) = (reverse rear, [])
maybe_mtf q = q
```

- move-to-front operation required when `front` is empty (obey invariant)
- single move-to-front operation may be expensive, but these operations are rare
- efficiency: n queue operations require at most $2n$ evaluation steps
- proving technique: amortized cost analysis, will be covered in course algorithms and data-structures

Abstraction Barrier of More Efficient Implementation

```
module Better_Queue(Queue, empty, isEmpty, dequeue, enqueue)
  where

type Queue a = ([a], [a])
...
empty :: Queue a
...
```

- since `type` is just an abbreviation:
  `empty :: ([a], [a])`
- since pairs and lists are visible, external users can completely inspect internal structure and create queues which are not permitted, e.g., `isEmpty ([], [4,3,2,1])` evaluates to `True`
- since `type` is just an abbreviation, in particular `Queue`'s are instances of `Eq`, `Show`, `Ord`, which might not be intended
- simple solution: hide represention in new datatype
  `data Queue a = Queue ([a], [a])`

Implementation with Separate Datatype

```haskell
module Data_Queue(Queue, empty, isEmpty, dequeue, enqueue)
  where

data Queue a = Queue ([a], [a])                -- new datatype

empty :: Queue a
empty = Queue ([], [])      -- wrap Queue constructor around

isEmpty :: Queue a -> Bool
isEmpty (Queue (f, _)) = null f -- unwrap Queue constructor

queue = Queue . maybe_mtf

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue (f, r)) = queue (f, x : r)

dequeue :: Queue a -> (a, Queue a)
dequeue (Queue ([], _)) = error "dequeue on empty queue"
dequeue (Queue (x : f, r)) = (x, queue (f, r))

maybe_mtf ([], r) = (reverse r, [])
maybe_mtf q = q
```

Newtype

```
data Queue a = Queue ([a], [a])

queue = Queue . maybe_mtf

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue (f, r)) = queue (f, x : r)
...
```

- always wrapping and unwrapping the `Queue` constructor has some efficiency penalty
- more efficient version to hide an implementation type: newtype
- syntax: `newtype TName tvars = CName typ`
  - only one constructor allowed
  - this constructor must have exactly one argument type
  - nearly equivalent to `data TName tvars = CName typ`,
    one difference: newtype is faster (`CName` won't be created at runtime)
- minimal change in implementation of queues
  - `newtype Queue a = Queue ([a], [a])` instead of
    `data Queue a = Queue ([a], [a])`

Summary

- names are bound inside-out
- conflicts of global names are resolved via qualifier
- larger programs can be structured in modules
- explicit export-lists to distinguish internal and external parts
- abstract datatypes: specify operations with their properties; introduces abstraction barriers that permit change of implementations
- newtype is efficient variant of `data` in case there is only one constructor with one argument
- example abstract datatypes: `Queue`, `Double`, `Char`, `Integer`, ...