Script: Functional data structures

Headline

Functional data structures in Haskell

Description

We look at functional programming techniques for implementing abstract data types and concrete data structures. In particular, we want to get a basic understanding of using familiar data types such as stacks or heaps without relying on side effects and thereby providing a form of persistence. To this end, we leverage so-called functional data structures which exploit (path) copying to achieve persistence while potentially leveraging lazy evaluation to achieve competitive performance (complexity).

Material

Functional data structures from **Ralf Laemmel**

Download slides

Functional data structures

Concepts

- Stack
- Abstract data type
- Lazy evaluation
- Functional data structure
- Binary search tree
- Skew heap
- Amortized analysis

Further reading

- Document:Handbook of data structures and applications
- Document:Okasaki96
- https://github.com/101companies/101repo/tree/master/concepts/Functional_data_structure

Metadata

- Course:Lambdas in Koblenz
- Script:Data modeling in Haskell

Concept: Amortized analysis

Headline

Analysis of algorithms based on sequences of operations

Metadata

- Complexity analysis
- http://en.wikipedia.org/wiki/Amortized_analysis
- https://www.cs.cmu.edu/~sleator/papers/adjusting-heaps.pdf

Concept: Abstract data type

Headline

A data type that does not reveal representation

Illustration

An abstract data type is usually just defined through the list of operations on the type, possibly enirched (formaly or informaly) by properties (invariants, pre- and postconditions).

Consider the following concrete data type for points:

```
data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)
```

Now suppose we want to hide the precise representation of points. In particular, we want to rule out that programmers can match and apply the constructor *Point*. The existing getters are sufficient to observe points without matching, but we need to provide some "public" means of constructing points.

```
mkPoint :: Int -> Int -> Point
mkPoint = Point
```

The idea is now to export *mkPoint*, but not the constructor, thereby making possible representation changes without changing any code that uses points. This is, of course, a trivial example, as the existing representation of points is probably quite appropriate, but see a more advanced illustration for an abstract data type Stack.

A complete module for an abstract data type for points may then look like this:

```
module Point(
  Point, -- constructor is NOT exported
  mkPoint,
  getX,
  getY
) where

data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)

mkPoint :: Int -> Int -> Point
mkPoint = Point
```

When defining an abstract data type, we take indeed the point of view that the representation and thus the implementation as such is not known or not to be looked at. Hence, ideally, the intended functionality should be described in some other way. For instance, we may describe the functionality by properties. For instance, in Haskell we may declare testable Technology:QuickCheck properties like this:

```
prop_getX :: Int -> Int -> Bool
prop_getX x y = getX (mkPoint x y) == x

prop_getY :: Int -> Int -> Bool
prop_getY x y = getY (mkPoint x y) == y
```

These properties describe the (trivial) correspondence between construction with *mkPoint* and observation with *getX* and *getY*. Logically, the first property says that for all given *x* and *y*, we can construct a point and we can retrieve *x* again from that point with *getX*.

Relationships

- An abstract data type is the opposite of a concrete data type.
- An abstract data type performs information hiding.

Metadata

- Vocabulary:Data
- http://en.wikipedia.org/wiki/Abstract_data_type
- Concrete data type
- Concept

Concept: Functional data structure

Headline

The specifically functional approach to the implementation ofdata structures

Illustration

See immutable lists as a simple example of afunctional data structure. See Document:Okasaki96 for a seminal resource (in fact, a PhD thesis) on the subject. See Document:Handbook of data structures and applications for a textbook with coverage of the subject. See Script:Functional data structures for a lecture on the subject.

Metadata

- Data structure
- Imperative data structure
- http://cstheory.stackexchange.com/questions/1539/whats-new-in-purely-functional-data-structures-since-okasaki

---

Concept: Lazy evaluation

Headline

Delay evaluation of an expression until its value is needed

Illustration

Lazy by definition

Lazy evaluation is either supported by the underlying programming language or it needs to be encoded by the programs. Let's start with illustrations in Haskell; this language's semantics is lazy by definition.

Consider the following expression and its evaluation:

```
> repeat 42
[42,42,42,42,42,42,42,42,42,42,42,42,42,42,...
```

That is, 42 is to be repeated an infinite number of times and all those 42s are to be collected in one list. It is not surprising that the evaluation of this expression never stops as witnessed by printing the infinite result forever. Laziness comes into play when such expressions are used in a way that they do not need to be fully evaluated.

For instance, let us compute the head of an infinite list:

```
> head $ repeat 42
42
```

Thus, the list of repeated 42s is never materialized; rather the infinite list is only computed up to the point needed for returning the result, i.e., the head of the list. Here is another example for exploiting laziness to compute on 'infinite' data:

```
> length $ take 42 $ repeat 42
42
```

That is, we compute the length of the list that holds the first 42 elements of the earlier infinite list of 42s. Here is yet another example:

```
> [1..] !! 41
42
```

That is, we retrieve the 42nd element (the 41st index) of the earlier list.

Lazy conditionals

Most languages are readily lazy in terms of the semantics of their conditionals such that the 'then' and 'else' branches are only evaluated or executed, if necessary. This specific form of

laziness is obviously important for programming, regardless of whether we face a language with lazy or strict evaluation. For instance, consider the following definition of factorial in Haskell:

```
-- A straightforward definition of factorial
factorial :: Integer -> Integer
factorial x =
  if x < 0
    then error "factorial arg error"
    else if x <= 1
      then 1
      else x * factorial (x-1)
```

Regardless of language, such a definition should not evaluate the recursive case, except when honored by the value of the condition. Thus, this style of recursive definition even works in a programming language with strict evaluation, .e.g, in Python:

```
# A straightforward definition of factorial
def factorial(x):
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        if x <= 1:
            return 1
        else:
            return x * factorial(x-1)
```

The difference between lazy and eager evaluation becomes quite clear, when we attempt a definition of 'if' as a function. In Haskell, we can actually define a function to mimic 'if' and use it in revising the recursive definition of factorial:

```
-- A re-definition of "if"
ifThenElse :: Bool -> x -> x -> x
ifThenElse True x   = x
ifThenElse False   x = x

-- Factorial re-defined to use user-defined if
factorial' :: Integer -> Integer
factorial' x =
  ifThenElse (x < 0)
    (error "factorial arg error")
    (ifThenElse (x <= 1)
      1
      (x * factorial' (x-1)))
```

The fact that this definition works depends on the lazy evaluation semantics of Haskell. The arguments of the function *ifThenElse* are only evaluated, when they are really needed. Let us attempt the same experiment in a language with eager evaluation semantics, e.g., Python:

```
# A troubled re-definition of "if"
def troubledIf(b,x1,x2):
    if b:
        return x1
    else:
        return x2

# Factorial re-defined to use user-defined if
def troubledFactorial(x):
```

```
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        return troubledIf(x<=1,1,x * troubledFactorial(x-1))
```

When exercising this definition, we get this sort of runtime error:

```
>>> troubledFactorial(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 21, in troubledFactorial
    raise RuntimeError('factorial arg error')
RuntimeError: factorial arg error
```

A quick analysis suggests that this runtime error arises from the fact that an application of the function 'generates' an infinite chain of recursive applications, thereby eventually leading to the application of the function to a negative number, which is intercepted by the precondition test of the function. Thus, the function *troubledIf* is clearly not lazy and it cannot be used in defining the factorial function.

Encoding laziness

One may encode laziness in a language with eager evaluation. To this end, each expressions, for which evaluation should be deferred, can be turned into a degenerated closure (lambda abstraction) such that the evaluation can be requested explicitly by a trivial application. Consider the following attempt at a user-defined 'if' in Python and its use in another attempt at the factorial function:

```
# A (properly) lazy re-definition of "if"
def lazyIf(b,x1,x2):
    if b:
        return x1(())
    else:
        return x2(())

# A definition of factorial using lazyIf
def lazyFactorial(x):
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        return lazyIf(x<=1,lambda  : 1, lambda  : x * lazyFactorial(x-1))
```

Thus, evaluation is requested explicitly by passing "()" (i.e., the empty tuple) to a "deferred" expression. When constructing a deferred expression, then we use a lambda abstraction with a

superfluous variable.

See Document:Okasaki96 for a profound discussion of data structures in a functional programming language while leveraging laziness for the benefit of efficiency.

Relationships

See the related concept of eager evaluation.

Synonyms (in a broad sense):

- Call-by-need evaluation
- Non-strict evaluation
- Laziness

Metadata

- http://en.wikipedia.org/wiki/Lazy evaluation
- Evaluation strategy
- Vocabulary:Programming
- Vocabulary:Programming languages

Concept: Stack

Headline

A last in, first out (LIFO) abstract data type

Illustration

A simple implementation of stacks (of ints) is shown here as a functional data structure in Language:Haskell:.

```
{-| A simple implementation of stacks in Haskell -}

module Stack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
data Stack = Empty | Push Int Stack

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Empty

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty Empty = True
isEmpty (Push _ _) = False

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push = Push

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top (Push x s) = x

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop (Push x s) = s

-- | Compute size of stack
size :: Stack -> Int
size Empty = 0
size (Push _ s) = 1 + size s
```

These stacks are immutable. The push operation does not modify the given stack; it returns a new stack which shares the argument stack possibly with other parts of the program. The pop operation does not modify the given stack; it returns a part of the argument stack. We refer to Document:Handbook of data structures and applications for a profound discussion of functional data structures including the stack example. The functions for operations top and pop, as given above, are partial because they are undefined for the empty stack.

There are also alternative illustrative Stack implementations available:

https://github.com/101companies/101repo/tree/master/concepts/Stack

Stacks as lists without information hiding

```
{-|

A leaky list-based implementation of stacks in Haskell.
That is, the representation type is not hidden.

-}

module LeakyListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
type Stack = [Int]

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = []

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty = null

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push = (:)

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop = tail

-- | Compute size of stack
size :: Stack -> Int
```

```
size = length
```

That is, stacks are represented as lists while the *Stack* type is simply defined as a type synonym to this end. This implementation does not enforce information hiding.

Stacks as lists with information hiding

```
{-|

An opaque list-based implementation of stacks in Haskell.
That is, the representation type is hidden.

-}

module OpaqueListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
newtype Stack = Stack { getStack :: [Int] }

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Stack []

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty = null . getStack

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push x s = Stack ( x : getStack s)

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head . getStack

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop = Stack . tail . getStack

-- | Compute size of stack
size :: Stack -> Int
size = length . getStack
```

As before, stacks are represented as lists, but the *Stack* type is defined as a newtype which hides the representation as its constructor is not exported.

Stack with length

```
{-|

An opaque list-based implementation of stacks in Haskell.
That is, the representation type is hidden.
The size of the stack is readily maintained.
Thus, the size can be returned with traversing the stack.

-}

module FastListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
data Stack = Stack { getStack :: [Int], getSize :: Int }

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Stack [] 0

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty = null . getStack

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push x s
  = Stack {
      getStack = x : getStack s,
      getSize = getSize s + 1
    }

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head . getStack

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop s
  = Stack {
      getStack = tail (getStack s),
      getSize = getSize s - 1
    }

-- | Compute size of stack
size :: Stack -> Int
size = getSize
```

As before, stacks are represented as lists and again this representation is hidden, but an additional data component for the size of the stack is maintained so that the size of a stack can be returned without traversing the stack.

An application of stacks

See Concept: Reverse_Polish_notation.

Metadata

- Abstract data type
- http://en.wikipedia.org/wiki/Stack (abstract data type)
- Vocabulary:Data

Document:

Handbook of data structures and applications

Headline

The Handbook of data structures and applications

Metadata

- Handbook
- http://www.e-reading-lib.org/bookreader.php/138822/Mehta - Handbook of Data Structures and Applications.pdf
- Data structure
- Functional data structure

Document: Okasaki96

Headline

Okasaki' PhD thesis on functional data structures

Metadata

- PhD thesis
- http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf
- Functional data structure

Concept: Binary search tree

Headline

A data structure supporting binary search

Illustration

See https://github.com/101companies/101repo/tree/master/concepts/Functional_data_structure.

Metadata

- Data structure
- http://en.wikipedia.org/wiki/Binary_search_tree

Concept: Skew heap

Headline

A data structure for self-adjusting heaps

Illustration

See https://github.com/101companies/101repo/tree/master/concepts/Functional_data_structure for an implementation of skew heaps as a functional data structure.

Metadata

- Data structure
- Heap
- http://en.wikipedia.org/wiki/Skew_heap
- http://www.cse.yorku.ca/~andy/courses/4101/lecture-notes/LN5.pdf

---