Novi Sad J. Math.
Vol. 31, No. 1, 2001, 141-149

NUMERICAL REPRESENTATIONS AS PURELY FUNCTIONAL DATA STRUCTURES

Mirjana Ivanović[1], Viktor Kunčak[2]

Abstract. This paper is concerned with design, implementation and verification of persistent purely functional data structures which are motivated by the representation of natural numbers using positional number systems. A new implementation of random-access list based on redundant segmented binary numbers is described. It uses 4 digits and an invariant which guarantees constant worst-case bounds for cons, head, and tail list operations as well as logarithmic time for lookup and update. The relationship of random-access list with positional number system is formalized and benefits of this analogy are demonstrated.

AMS Mathematics Subject Classification (1991): 68N18, 68Q65

Key words and phrases: functional data structures, numerical representations, Haskell

1. Introduction

Study of data structures in the context of purely functional programming languages is important both for improving efficiency of functional programs and for exploring issues in foundation of data structures. New techniques are needed to analyze persistent data structures, which are naturally favored by purely functional languages, yet have advantages even in imperative settings.

Implementing data structures in functional programming languages makes them closer to their specification, facilitating formal development of operations. In this way, the implementation can be derived along with the proof of its correctness and properties of data structures can be rigorously studied.

In this paper a class of purely functional data structures termed *numerical representations* is explored. Contributions of this paper are:

- formalization of analogy with the number system;
- implementation and correctness proof for a segmented representation based on 4 instead of 5 digits, using a new invariant.

[1]Faculty of Science, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia , e-mail: mira@unsim.ns.ac.yu

[2]Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA, e-mail: vkuncak@mit.edu

On the operational side, using a pure functional programming language makes data structures *persistent* ([5]). For many abstract data types such as lists, queues, trees, and heaps both persistent and imperative implementations exist.

Implementations in this paper are written in Haskell, a non-strict, purely functional programming language. Programs were tested using Hugs [3] environment. Notation of multiparameter type classes and instance declarations is used, but it is not central to this approach.

2 Random-Access List

This section introduces the notion of random-access list (RAL). The signature of this data structure is presented as a Haskell type class and a minimal implementation is given.

Motivation. RAL is a data structure which implements both list and array interfaces. Elements can be inserted in the front, but also the i-th element can be replaced or retrieved efficiently. In this respect, random-access lists have similar functionality as imperative arrays. Unlike static arrays, however, they can grow arbitrarily. Even if implementations of vectors can be made that dynamically expand and shrink, they are not persistent since the update operation destroys the previous version of data structure. Hence RAL are the best choice for persistent lists with efficient indexing.

List interface. The list interface can be described by the following multiparameter type class.

```
class Lst r a where
  empty   :: r a
  cons    :: a -> r a -> r a
  isEmpty :: r a -> Bool
  head    :: r a -> a
  tail    :: r a -> r a
```

Here `r` is a type constructor, so `r a` is a list of elements of type `a`. The class introduces two abstract list constructors `empty` (make empty list) and `cons` (add element to the front of the list), as well as destructors: `isEmpty` to test whether the list is empty, and `head` and `tail` to access head and tail of a nonempty list.

Array interface. The array interface is given by the following class. Function `size` is introduced since array can grow and shrink over the time.

```
class Arr r a where
  size   :: r a -> Int
  lookup :: Int -> r a -> a
  update :: a -> Int -> r a -> r a
```

The definition of RAL signature is just

```
class (Lst r a, Arr r a) => RandomAccessList r a
instance (Lst r a, Arr r a) => RandomAccessList r a
```

Minimal implementation. The usual implementation of the list is obtained by treating `empty` and `cons` as free algebra generators.

```
data List a = Nil | Cons {headL :: a, tailL :: List a}
instance Lst List a where
  empty = Nil
  cons  = Cons
  isEmpty lst = case lst of
                  Nil -> True
                  _   -> False
  head = headL
  tail = tailL
```

This definition is identical, up to syntactic sugar, to the built-in implementation of lists in Haskell.

In this implementation efficiency problems arise with array interface operations. The best that can be achieved is linear complexity for `lookup` and `update`.

```
instance Arr List a where
  size Nil = 0
  size (Cons _ lst) = 1 + size lst

  lookup 0     (Cons a as) = a
  lookup (n+1) (Cons a as) = lookup n as

  update x 0     (Cons a as) = Cons x as
  update x (n+1) (Cons a as) = Cons a (update x n as)
```

In the sections to follow RAL implementations will be introduced with logarithmic `update` and `lookup` operations. Due to its simplicity, the implementation in this section can be used for correctness verification of more complex implementations.

Simple Binary Random-Access List. This section presents the analogy with positional number systems, which is the essential idea of numerical representations. The RAL based on ordinary binary number system is used to demonstrate advantages of this approach.

Binary numbers representation can be written in Haskell as follows.

```
data Digit = Zero | One
type BinNum = [Digit]
```

The following simple definitions of functions for incrementing (`inc`) and decrementing (`dec`) binary numbers will be used as abstract descriptions of the RAL operations `cons` and `tail`.

```
inc [] = [One]
inc (Zero:ds) = One:ds
inc (One:ds) = Zero:inc ds
```

```
dec [One] = []
dec (One:ds) = Zero:ds
dec (Zero:ds) = One:dec ds
```

Deriving `Lst` implementation. While natural numbers from the previous subsection are lists of digits, the RAL based on binary numbers is a list of "tree digits".

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
data TreeDigit a = ZeroT | OneT (Tree a)
type SimpleRAL a = [TreeDigit a]
```

`OneT` tree digit holds a complete binary leaf tree. Complete binary leaf tree of height h has 2^h elements. The i-th tree digit in the RAL holds 2^i elements, which justifies analogy with the binary number system.

To formalize the analogy between `BinNum` and `SimpleRAL`, functions `abst` and `mabst` are introduced. `abst` just throws away the tree, and `mabst` applies `abst` to all elements of the list.

```
abst :: TreeDigit a -> Digit
abst ZeroT    = Zero
abst (OneT _) = One
mabst :: SimpleRAL a -> BinNum
mabst = map abst
```

These functions can be used to guide the derivation of RAL operations. The `cons` operation on trees is defined as follows.

```
consR :: a -> SimpleRAL a -> SimpleRAL a
consR a = insTree (Leaf a)
insTree :: Tree a -> SimpleRAL a -> SimpleRAL a
```

The analogy between numbers and RAL is given by the following equation:

```
mabst . insTree t = inc . mabst
```

Here . denotes function composition. By expanding this specification, a pattern for the definition of `insTree` is obtained. Both sides of the equation have the type `SimpleRAL a -> BinNum`, so that the desired equation becomes

```
mabst (insTree t ts) = inc (mabst ts)
```

for all trees `t` and all sequences of tree digits `ts`.

In the similar vein, operation `unconsTree` can be derived from `dec` operation on the binary numbers. While the type of `insTree` was isomorphic to `(Tree a, SimpleRAL a) -> SimpleRAL a`, the type of `unconsTree` is

```
unconsTree :: SimpleRAL a -> (Tree a, SimpleRAL a)
```

This operation is used to define the RAL **head** and **tail** operations.

```
headR ral = let (Leaf a, _) = unconsTree ral in a
tailR ral = let (_, tl)     = unconsTree ral in tl
```

The specification in this case is

```
mabst . snd .. unconsTree = dec . mabst
```

where **snd (x,y) = y**. The derivation of **unconsTree** would proceed again by induction on the structure of a RAL.

Writing Arr implementation. It remains to write **sizeR**, **lookupR**, and **updateR** functions for the RAL. The implementation of **sizeR** is simple and **mabst** makes it even simpler.

```
sizeR = binVal . mabst
```

Here **binVal** calculates the value of a binary number.

```
binVal = foldr op 0 where
  op d r = digitVal d + 2*r
digitVal Zero = 0
digitVal One  = 1
```

Since **mabst = map abst**, a simple Haskell implementation would execute this definition of **sizeR** by creating an intermediate list. More efficient version would be obtained if **abst** were propagated to the definition of **digitVal**.

Implementations of **lookup** and **update** are straightforward once the linear order is imposed on RAL elements. In the list of trees, elements in earlier trees come first. Inside the tree, leaves are ordered left to right.

```
lookupR :: Int -> SimpleRAL a -> a
lookupR = lookup1 1
lookup1 sz i (ZeroT:rl) = lookup1 (2*sz) i rl
lookup1 sz i (OneT t:rl)
  | i < sz    = lookupTree sz i t
  | otherwise = lookup1 (2*sz) (i-sz) rl

lookupTree sz 0 (Leaf x) = x
lookupTree sz i (Node t1 t2)
  | i < sz2   = lookupTree sz2 i t1
  | otherwise = lookupTree sz2 (i-sz2) t2
  where sz2 = sz `div` 2
```

This completes the implementation of RAL based on simple binary number system. The main purpose of this section was to demonstrate the benefits of using analogy with positional number systems. The RAL implementation derived here has $O(n)$ worst-case complexity for `cons` and `tail`. This corresponds to linear worst-case complexity for `inc` and `dec`, as in `inc [1,1,1,1,1] = [0,0,0,0,0,1]` and `dec [0,0,0,0,1] = [1,1,1,1]`. In general, incrementing $2^k - 1$ takes about k steps, as does decrementing 2^{k+1}. Although cases with such "cascading carries" and "cascading borrows" are rare and can be amortized in non-persistent usage of data structure ([2]), this is *not* true for persistent usage ([5]) of data structures based on binary numbers.

3. Random-Access List via Recursive Slowdown

This section presents implementation of a random-access list with $O(1)$ worst-case bounds on `cons`, `head` and `tail` operations. Moreover, `lookup i` and `update x i` will have $O(\log i)$ worst-case complexity. The implementation uses 4 instead of 5 digits ([5]) and relies on slightly different invariant.

The relevance of the analogy with the number system should become obvious here: invariants which are the essence of this implementation can all be proved considering the number system alone.

Segmented redundant binary numbers. The motivation behind segmented redundant binary numbers is to avoid cascading carries in `inc` and cascading borrows in `dec`. To achieve this, additional digits 2 and 3 are introduced. Positional binary system is still used. However, the representation of the number is not unique any more and reflects previous applications of `inc` and `dec`.

Introducing new digits 2 and 3 does not solve the problem by itself. Cascading carries could now appear in cases such as `[3,3,3,3,3]`. What is needed is a constraint on the digit sequence which would eliminate such cases. The constraint chosen here is that every digit three is preceded by digit zero or one, possibly followed by a list of two-s. Analogously, zero is preceded by two or three, possibly followed by a list of one-s. This is the *invariant* that will hold for representation of the number 0, and which `inc` and `dec` need to preserve. The invariant can be described by two regular expressions:

(A) $((0+1)2^*3+0+1+2)^*$

(B) $((3+2)1^*0+3+2+1)^*$

The symmetry between digits is apparent in the invariants: replacing the digit d by $3-d$ in (A) yields (B) and vice versa.

In order to check invariants (A) and (B), the ability to skip over a sequence of one digits and two digits of arbitrary length is needed. Therefore, consecutive digits are grouped into the list, yielding the following data structure.

```
data Digit = Zero | Ones Int | Twos Int | Three
data SegNum = [Digit]
```

To make sure that all consecutive ones and twos are in one group, the functions `ones` and `twos` are used.

```
ones :: Int -> SegNum -> SegNum
```

Incrementing a number is done in two steps: incrementing the first digit by `simpleInc`, and restoring the invariant by `fixInc`.

```
inc :: SegNum -> SegNum
inc = fixInc . simpleInc
```

In this case `inc` and `dec` run in $O(1)$ time. This will lead directly to $O(1)$ implementation of `cons` and `tail` for RAL.

RAL based on segmented redundant binary numbers. This subsection extends `inc` and `dec` operations on the number system of previous subsection to `cons`, `tail`, and `head` operations in a random-access list. The underlying number system is more complex.

The first step is to extend the data structure. Each digit holds the number of trees equal to its value. Sequences of digits are represented by lists of (pairs of) trees.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
data TreeDigit a = ZeroT
                 | OnesT  [Tree a]
                 | TwosT  [(Tree a,Tree a)]
                 | ThreeT (Tree a, Tree a, Tree a)
type SegmenRAL a = [TreeDigit a]
```

Auxiliary functions that keep consecutive one's and two's together take lists of trees as arguments. Definition of `consR` should come as no surprise given `consR` for `SimpleRAL` and `inc`. Taking into account the order of elements one obtains the following definition.

```
consR a = fixIns . simpleIns (Leaf a)
```

Operations `headR` and `tailR` are implemented using `simpleUncons`, which generalizes `simpleDec`, and `fixUncons`, which generalizes `fixDec`.

```
headR :: SegmenRAL a -> a
headR ral = a where (Leaf a, _) = simpleUncons ral

tailR :: SegmenRAL a -> SegmenRAL a
tailR = fixUncons . snd . simpleUncons
```

This completes the implementation of `Lst` interface for RAL. As in Section 2, the relationship with the number system could be formalized by `abst` and `mabst`.

```
abst :: TreeDigit a -> Digit
abst ZeroT      = Zero
abst (OnesT ts) = Ones (length ts)
abst (TwosT ts) = Twos (length ts)
abst ThreeT     = Three
mabst :: SegmenRAL -> SegNum
mabst = map abst
```

Implementation of the operations `lookup` and `update` of the `Arr` interface requires some work, but no new insights. The order of elements in `SegmenRAL` data structure corresponds to their order in standard printed representation.

Worst-case bounds. Worst-case time complexity bounds for the resulting random-access list are given in the following table.

operation	worst-case complexity
`consR a ral`	$O(1)$
`headR a ral`	$O(1)$
`tailR a ral`	$O(1)$
`lookupR i ral`	$O(\log i)$
`updateR i a ral`	$O(\log i)$

Constant times for `consR`, `headR`, and `tailR` are obvious from their definitions.

4. Conclusions and future work

The random-access list presented in this paper is among the most efficient *persistent* implementations that support *both* list and array abstract data types. In [5], several random-access list implementations are presented. Among them, random-access list based on *skew* number systems deserves special attention because it is efficient and simple. Its potential drawback is that `lookup i` and `update i a` can take $O(\log n)$, where n is total number of list elements, compared to $O(\log i)$ for segmented representation. The $O(\log i)$ bound is also achieved by another implementation from [5], which essentially relies upon laziness. This makes it unsuitable for strict programming languages and makes complexity analysis more involved. In addition, the resulting bounds are amortized and not worst-case. *Scheduling* technique is needed to achieve worst-case bounds, which further complicates the implementation. For this reason, segmented representation was chosen here. It was shown that the desired effect can be achieved using digits 0, 1, 2, and 3 instead of 5 digits as suggested in [5].

Experience in implementations shows that purely functional languages are an excellent vehicle for the development of new persistent data structures. It is worth stressing again that persistent data structures are not specific for functional programming languages. Both persistent and mutable data structures can be used in both functional and imperative programming paradigms. Although persistence requirements may seem constraining, it would not be the

first time that a more controlled use of language features resulted in a better programming practice.

References

[1] Bird, R., Introduction to functional programming using Haskell, 2nd Edn. Prentice Hall, 1998.

[2] Cormen, H., Leiserson, C.E., Rivest, R.L., Introduction to Algorithms, MIT, 1990.

[3] Jones, M.P., Peterson, J.C., Hugs98 User Manual, Revised Version: September 1999, **http://haskell.org/hugs**

[4] Peyton
Jones, S.L., Hughes, J., Haskell 98: A Non-strict, Purely Functional Language, February 1999, language report available from **http://haskell.org/report**.

[5] Okasaki, C., Purely Functional Data Structures, CUP, 1998.

[6] Pippenger, N., Pure versus Impure Lisp. ACM Transactions on Programming Languages and Systems, Vol. 19, No. 2, March 1997, 223-238.

[7] Peyton Jones, S.L., Wadler, P., Imperative Functional Programming, ACM Symposium on Principles of Programing Languages (POPL), Charleston, January 1993, 71-84.