INFORMATICA, 2002, Vol. 13, No. 2, 163–176

Numerical Representations as Purely Functional Data Structures: a New Approach

Mirjana IVANOVIĆ
Faculty of Science and Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: mira@im.ns.ac.yu

Viktor KUNČAK
Laboratory for Computer Science, Massachusetts Institute of Technology
Cambridge, MA02139
e-mail: vkuncak@mit.edu

Received: May 2001

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.

Key words: data structures, purely functional language, random-accesss list, program derivation, recursive slowdown.

1. Introduction

Studying data structures in the context of purely functional programming languages is important (Aditya, 1995) both for improving efficiency of functional programs and for exploring issues in foundations of data structures (Reid, 1989; Bird and Wadler, 1988). 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 (Voß, 1985). In this way, the implementation can be derived along with the proof of its correctness and properties of data structures can be rigorously studied.

Functional programming languages abstract from the issues of memory management and references, which results in clear, concise and easy to debug programs (Bird, 1998; Hughes, 1989; Turner, 1982). This makes them particularly suitable for developing and experimenting with new data structures. This is a consequence of uniform treatment for all values. Data structures are just values of algebraic data types, and their use and modification in functional style is explicit.

In this paper a class of purely functional data structures termed *numerical representations* is explored. The discussion here is motivated by a chapter in (Okasaki, 1998). Contributions of this paper are:

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

On the operational side, using a pure functional programming language makes data structures *persistent* (Okasaki, 1998). Persistent data structures remain available after updates have been performed, so new version of data structure coexists with the original one.[1] Persistent data structures are an important subclass of data structures. For many abstract data types such as lists, queues, trees, and heaps both persistent and imperative implementations exist (Pippenger, 1997; Wadler, 1995; Peyton Jones and Wadler, 1993). The advantages of persistent implementations are:

- they can be used in purely functional programming languages without language extensions;
- they can also be used in imperative languages (especially those with garbage collection support), where they avoid expensive copy operations;
- being read-only, they offer greater potential for performance improvements via caching and parallelization;
- reasoning about them is simpler.

Their potential disadvantage is less efficient memory use if only one version of data structure is needed. Persistent implementations of some abstract data types tend to be more complex that imperative implementations and in some cases have worse asymptotic time and space bounds.

Implementations in this paper are written in Haskell (Peyton Jones and Hughes, 1999), a non-strict, purely functional programming language. Programs were tested using Hugs environment (Jones and Peterson, 1999). Notation of multiparameter type classes and instance declarations is used, but it is not central to this approach.

The paper is organized as follows. In the second section the notion of random-access list is introduced. In the third section the analogy with positional number systems, essential for numerical representation is presented. New approach to implementation of random-access list is described in the section four. Conclusion and further work are given in the last section.

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.

[1] This is not to be confused with persistence as a language capability for storing values on external storage for later use.

2.1. *Motivation*

RAL is a data structure which implements both list and array interfaces. Elements can be inserted in the front, but also 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 (Trinder, 1989). Hence RAL are the best choice for persistent lists with efficient indexing.

2.2. *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 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.

2.3. *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
```

2.4. *A Minimal Implementation*

The usual implementation of 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 following sections 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.

3. Simple Binary Random-Access List

This section presents the analogy with positional number systems which is the essential idea of numerical representations. RAL based on ordinary binary number system is used to demonstrate advantages of this approach. The implementation is similar to the one in (Okasaki, 1998), but the presentation here is slightly more in the spirit of program derivation.

3.1. *Binary Numbers*

In binary number system a natural number is represented as a list of ones and zeros. The weight of the i-th digit is 2^i, so the value of binary digit sequence $a_0 a_1 \ldots a_n$ is $a_0 \cdot 2^0 + a_1 \cdot 2^1 + \cdots + a_n \cdot 2^n$. Note that here (contrary to the usual practice) the least significant digit is written first.

This 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 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
```

Operations `inc` and `dec` preserve the absence of trailing zeros in the digit sequence, which is easy to verify by induction on the length of the sequence.

3.2. *Deriving* `Lst` *Implementation*

While natural numbers from previous subsection are lists of digits, 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-the 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 the desired equation becomes

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

for all trees `t` and all sequences of tree digits `ts`. The informal derivation of `insTree` proceeds by case analysis.

Case `ts=[]` The right hand side evaluates to `[One]`. By definition of `mabst`, it must be `insTree t [] = t1` where `t1` is some tree. The most natural choice `t=t1` turns out to be the right one. Hence

```
insTree t [] = [OneT t]
```

Case `ts=ZeroT:ts1` The right hand side evaluates to `One:mabst ts1`, or, by definition of `mabst`, `mabst (OneT t1:ts1)`. One way to satisfy the equation

```
mabst (insTree t (ZeroT:ts1)) = mabst (OneT t1:ts1)
```

is to make arguments of `mabst` equal. By taking `t=t1`, this case becomes

```
insTree t (ZeroT:ts1) = OneT t:ts1
```

Case `ts=OneT t1:ts1` The right hand side can now be written in the form `Zero:inc (mabst ts1)`. Assuming the equation holds for `ts1`, this becomes `Zero:mabst (insTree t2 ts1)` for some tree `t2`, which equals `mabst (ZeroT:insTree t2 ts1)`. This can again be satisfied by stripping off `mabst`, giving

```
insTree t (OneT t1:ts1) = ZeroT:insTree t2 ts1
```

In order to keep all the elements it is reasonable to instantiate the free variable `t2` by putting `t2=Node t t1`, which results in the final case

```
insTree t (OneT t1:ts1) = ZeroT:insTree (Node t t1) ts1}
```

The three cases just derived make up a complete definition of `insTree`. In the similar vein, operation `unconsTree` can be derived from `dec` operation on 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 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.

The operations `consR`, `headR`, and `tailR` accompanied by definitions `emptyR = []` and `isEmptyR ral = (ral==[])` make up the implementation of `Lst` interface for this RAL. Due to a restriction on type synonyms, writing an actual instance declaration for `Lst` multiparameter class would require the use of `newtype` in the definition of `SimpleRAL` which would clutter the code with application of trivial type constructor and destructors. Instead, function implementations here are simply suffixed by letter `R`.

3.3. *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

updateR :: Int -> a -> SimpleRAL a -> SimpleRAL a
updateR = update1 1
update1 sz i x (ZeroT:rl) = ZeroT : update1 sz i x rl
update1 sz i x (OneT t:rl)
  | i < sz    = OneT (updateTree sz i x t) : rl
  | otherwise = OneT t : update1 sz (i-sz) x rl

updateTree sz 0 x (Leaf _) = Leaf x
updateTree sz i x (Node t1 t2)
  | i < sz2    = Node (updateTree sz2 i x t1) t2
  | otherwise  = Node t1 (updateTree sz2 (i-sz2) x 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 [0,0,0,0,0,1] = [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 (Cormen et al., 1990), this is *not* true for persistent usage (Okasaki, 1998) of data structures based on binary numbers.

4. Random-Access List via Recursive Slowdown

This section presents an implementation of random-access list with $\mathrm{O}(1)$ worst-case bounds on `cons`, `head` and `tail` operations. Moreover, `lookup i` and `update x i` will have $\mathrm{O}(\log i)$ worst-case complexity. The implementation is similar to the one suggested in (Okasaki, 1998), but uses 4 instead of 5 digits and relies on slightly different invariant.

The relevance of analogy with number system should become obvious here: invariants which are the essence this implementation can all be proved considering the number system alone. Then it becomes easy to extend the implementation to RAL.

4.1. *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 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 invariants: replacing 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 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, functions `ones` and `twos` are used instead of constructors Ones and Twos.

```
ones :: Int -> SegNum -> SegNum
ones 0 ds = ds
ones i (Ones k:ds) = Ones (i+k) : ds
ones i ds = Ones i : ds
twos :: Int -> SegNum -> SegNum
twos 0 ds = ds
twos i (Twos k:ds) = Twos (i+k) : ds
twos i ds = Twos i : ds
```

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

simpleInc :: SegNum -> SegNum
simpleInc [] = [Ones 1]
simpleInc (Zero:ds)   = ones 1 ds  - only for fixInc
simpleInc (Ones i:ds) = twos 1 (ones (i-1) ds)
simpleInc (Twos i:ds) = Three:twos (i-1) ds

fixInc :: SegNum -> SegNum
fixInc (Twos i:Three:ds) = Twos i:ones 1 (simpleInc ds)
fixInc (Three:ds) = ones 1 (simpleInc ds)
fixInc ds = ds
```

Note that `simpleInc` is well defined. First, (A) guarantees that the first digit is never Three, so `simpleInc` in `inc` is well-defined. Next, if the argument of `simpleInc` in `fixInc` had a leading Three, it would mean that (A) was violated in the original digit sequence.

(A) can be violated by turning One into Two in front of Three or by turning Two into Three. Both of these cases are dealt with by `fixInc`. Although `fixInc` may call `simpleInc` again creating another Two or Three, `simpleInc ds` is preceded by One, so (A) is not violated any more.

(B) is not violated by `simpleInc`, so the only danger is that `fixInc` turns a Three into One in front of a sequence of Ones and a Zero. But in this case `simpleInc` increments Zero or One so the invariant still holds.

Hence `inc` preserves both invariants. The definition and proof for `dec` are analogous.

```
simpleDec :: SegNum -> SegNum
simpleDec [Ones 1] = []
simpleDec (Ones i:ds) = Zero:ones (i-1) ds
simpleDec (Twos i:ds) = ones 1 (twos (i-1) ds)
simpleDec (Three:ds) = twos 1 ds

fixDec :: SegNum -> SegNum
fixDec (Ones i:Zero:ds) = Ones i:twos 1 (simpleDec ds)
fixDec (Zero:ds) = twos 1 (simpleDec ds)
fixDec ds = ds
```

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

4.2. *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 random-access list. The extension is similar to one in Section 3, but 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 Ones and Twos together take lists of trees as arguments.

```
onesT :: [Tree a] -> SegmenRAL a -> SegmenRAL a
onesT [] ds = ds
onesT ts (OnesT os:ds) = OnesT (ts++os) : ds
onesT ts ds = OnesT ts : ds

twosT :: [(Tree a,Tree a)] -> SegmenRAL a -> SegmenRAL a
twosT [] ds = ds
twosT ts (TwosT tws:ds) = TwosT (ts++tws) : ds
twosT ts ds = TwosT ts : ds
```

Definition of `consR` should come as no surprise given `consR` for `SimpleRAL` of Section 3 and `inc` of previous subsection. Taking into account the order of elements leads to the following definition.

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

simpleIns :: Tree a -> SegmenRAL a -> SegmenRAL a
simpleIns t [] = [OnesT [t]]
simpleIns t (ZeroT:ds) = onesT [t] ds
simpleIns t (OnesT (t1:ts):ds) = twosT [(t,t1)] (onesT ts ds)
simpleIns t (TwosT ((t1,t2):tws):ds)
  = ThreeT (t,t1,t2) : twosT tws ds

fixIns :: SegmenRAL a -> SegmenRAL a
fixIns (TwosT tws:ThreeT (t1,t2,t3):ds)
  = TwosT tws:onesT [t1] (simpleIns (Node t2 t3) ds)
fixIns (ThreeT (t1,t2,t3):ds)
  = onesT [t1] (simpleIns (Node t2 t3) ds)
fixIns ds = ds
```

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

simpleUncons :: SegmenRAL a -> (Tree a, SegmenRAL a)
simpleUncons [OnesT [t]] = (t, [])
simpleUncons (OnesT (t:ts):ds) = (t, ZeroT:onesT ts ds)
simpleUncons (TwosT ((t1,t2):ts):ds)
  = (t1, onesT [t2] (twosT ts ds))
simpleUncons (ThreeT (t1,t2,t3):ds) = (t1, twosT [(t2,t3)] ds)

fixUncons :: SegmenRAL a -> SegmenRAL a
fixUncons (OnesT ts:ZeroT:ds) = OnesT ts:twosT [(t1,t2)] ds1
  where (Node t1 t2, ds1) = simpleUncons ds
fixUncons (ZeroT:ds) = twosT [(t1,t2)] ds1
  where (Node t1 t2, ds1) = simpleUncons ds
fixUncons ds = ds
```

This completes the implementation of `Lst` interface for RAL. As in Section 3 the relationship with 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
```

The following equations are then easy to verify.

1. `mabst . onesT ts = ones (length ts) . mabst`
2. `mabst . twosT ts = twos (length ts) . mabst`
3. `mabst . simpleIns t = simpleInc . mabst`
4. `mabst . fixIns = fixInc . mabst`
5. `mabst . consR a = inc . mabst`
6. `mabst . snd . simpleUncons = simpleDec . mabst`
7. `mabst . fixUncons = fixDec . mabst`
8. `mabst . tailR = dec . mabst`

In particular, 5 follows immediately from 3 and 4, and 8 follows from 6 and 7.

Implementation of 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. The definition of `lookupR` is bellow and the structure of `updateR` implementation is analogous.

```
lookupR = lookupList 1
lookupList :: Int -> Int -> SegmenRAL a -> a
lookupList sz i (ZeroT:ds)    = lookupList (2*sz) i ds
lookupList sz i (OnesT ts:ds) = lookupOnes sz i ts ds
lookupList sz i (TwosT ts:ds) = lookupTwos sz i ts ds
```

```
lookupList sz i (ThreeT (t1,t2,t3):ds)
  | i < sz      = lookupTree sz i        t1
  | i < 2*sz    = lookupTree sz (i-sz)   t2
  | i < 3*sz    = lookupTree sz (i-2*sz) t3
  | otherwise   = lookupList (2*sz) (i-3*sz) ds

lookupOnes sz i [] ds = lookupList sz i ds
lookupOnes sz i (t:ts) ds | i < sz    = lookupTree sz i t
  | otherwise = lookupOnes (2*sz) (i-sz) ts ds
lookupTwos sz i [] ds = lookupList sz i ds
lookupTwos sz i ((t1,t2):ts) ds
  | i < sz      = lookupTree sz i t1
  | i < 2*sz    = lookupTree sz (i-sz) t2
  | otherwise   = lookupTwos (2*sz) (i-2*sz) ts ds

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
```

4.3. *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.

Logarithmic time bound for `lookupR i` and `updateR i a` follows from following reasoning. Let the i-th node be located in k-th TreeDigit of the random-access list. According to invariant (B), every Zero digit is preceded by Two or Three. Therefore preceding $k-1$ digits contain at least $k-1$ trees. There are up to 3 trees in a tree digit, so there are at least $(k-1)/3$ different tree sizes with at least $3(2^{(k-1)/3}-1)$ elements. Therefore $i \geqslant 3(2^{(k-1)/3}-1)$, so k is a logarithmic function of i. Search for i-th element proceeds through first k elements of RAL, and through the k-tree whose depth is k. The number of steps in `lookupR` is bounded by a linear function of k, so it is a logarithmic function of i. Similar argument holds for `updateR`.

5. Conclusions and Future Work

Random-access list presented in this paper is among the most efficient *persistent* implementations that support *both* list and array abstract data types. In (Okasaki, 1998) 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 $\mathrm{O}(\log n)$ where n is total number of list elements, compared to $\mathrm{O}(\log i)$ for segmented representation. The $\mathrm{O}(\log i)$ bound is also achieved by another implementation from (Okasaki, 1998), which essentially relies on 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 (Okasaki, 1998).

The analogy with number system proved to be extremely useful on both intuitive and formal level of reasoning. Full verification of implementation was not done, but no serious difficulties are expected in this direction. The ability to use the same language both for stating properties and writing efficient implementations is an important advantage itself. It allows application of program transformation techniques which promise to improve the quality of programming process.

This experience shows that purely functional languages are an excellent vehicle for 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 better programming practice.

References

Aditya, S. (1995). Functional encapsulation and type reconstruction in a strongly-typed. Polymorphic language. *PhD Thesis*. MIT.

Bird, R., Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.

Bird, R. (1998). *Introduction to Unctional Programming Using Haskell.* 2nd Edn. Prentice Hall.

Cormen, H., Leiserson, C. E., Rivest, R.L. (1990). *Introduction to Algorithms.* MIT.

Hughes, J. (1989). Why functional programming matters. *Computer Journal*, **32**(2).

Jones, M.P., Peterson, J.C. (1999). *Hugs98 User Manual*. Revised version. `http://haskell.org/hugs`.

Okasaki, C. (1998). *Purely Functional Data Structures.* CUP.

Peyton Jones, S.L., Wadler, P. (1993). Imperative Functional Programming. In *ACM Symposium on Principles of Programing Languages (POPL)*. Charleston. pp.71–84.

Peyton Jones, S.L., Hughes J. (1999). *Haskell 98: A Non-strict, Purely Functional Language.* Language report available from `http://haskell.org/report`.

Pippenger, N. (1997). Pure versus Impure Lisp. In *ACM Transactions on Programming Languages and Systems*. Vol. 19(2), pp. 223–238.

Reid, A. (1989). Designing Data Structures. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag.

Trinder, P. (1989). Referentially transparent database languages. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag.

Turner, D. A. (1982). Recursion Equations as a Programming Language. In J. Darlington, P. Henderson, D. A. Turner (Eds.), *Functional Programming and its Applications*. Cambridge University Press. Cambridge.

Voß, A. (1985). Algebraic specifications in an integrated software development and verification system. *PhD Thesis*. University of Kaiserslautern.

Wadler, P. (1995). How to declare imperative. In J. Loyd (Ed.) *International Logic Programming Symposium*. MIT Press.

M. Ivanović received MSc degree in Computer Science from Novi Sad University in 1988 and PhD degree in Computer Science from the same university in 1992. Presently she is an associate professor at Institute of Mathematics and Computer Science, Faculty of Science, University of Novi Sad. Her scientific interests include programming languages, agent oriented methodology, software engineering and comiplers.

V. Kunčak recieved his BSc degree in Computer Science from University of Novi Sad in 2000 with Best University Student Award. He is currently graduate student in Laboratory for computer science of Massachusetts Institute of Technology. His main interests include program analysis and verification, lambda calculus, and programming language implementation and design.

Skaitmenų grupės kaip išskirtinai funkcinės duomenų struktūros: naujas požiūris

Mirjana IVANOVIĆ, Viktor KUNČAK

Straipsnyje nagrinėjamos nuolat saugomų išskirtinai funkcinių duomenų struktūrų projektavimo, realizavimo ir verifikavimo problemos. Šias problemas siūloma spręsti pasinaudojant natūrinių skaičių vaizdavimo pozicinėse skaičiavimo sistemose būdu. Straipsnyje taip pat yra pasiūlytas naujas atsitiktinės prieities sąrašo realizavimo būdas, grindžiamas dvejetainių skaitmenų pasikartojančių grupių panaudojimu. Naudojami 4 skaitmenys ir invariantinė dalis, šitaip užtikrinant, kad darbo su sąrašinėmis struktūromis operacijoms bus garantuotos fiksuotos blogiausiojo atvejo ribos ir logaritminės laiko sąnaudos paieškos bei atnaujinimo operacijoms. Atsitiktinės prieities sąrašo sąryšis su pozicine skaičiavimo sistema yra formalizuotas, straipsnyje parodyta, kokius privalumus duoda šitokia analogija.