rh163 : 0

A Logarithmic Implementation of Flexible Arrays

Rob R. Hoogerwoord

Eindhoven University of Technology, department of Mathematics and Computing Science,
postbus 513, 5600 MB Eindhoven, The Netherlands

Abstract. In this paper we derive an implementation of so-called *flexible arrays*; a flexible array is an array whose size can be changed by adding or removing elements at either end. By representing flexible arrays by so-called *Braun trees*, we are able to implement all array operations with logarithmic —in the size of the array— time complexity.
Braun trees can be conveniently defined in a recursive way. Therefore, we use functional programming to derive (recursive) definitions for the functions representing the array operations. Subsequently, we use these definitions to derive (iterative) sequential implementations.

0 Introduction

A flexible array is an array the size of which can be changed by adding or removing elements at either end. In 1983 W. Braun and M. Rem designed an implementation of flexible arrays by means of balanced binary trees, which are used in such a way that all array operations can be performed in logarithmic time. Examples of programs in which flexible arrays are used can be found in [1].

The original presentation of this design by Braun and Rem is, however, rather complicated [0]. In this paper we use functional programming to derive this implementation in a more straightforward way. We do so in three steps. First, the binary trees used to represent arrays are defined as a recursive data type. Second, we derive recursive definitions for the functions implementing the array operations; this is relatively easy. Finally, we use these definitions to derive iterative sequential implementations of the array operations, where the Braun trees are represented by means of nodes linked together by pointers.

1 Functional Programs

1.0 Specifications

For the sake of simplicity of presentation, we assume that all (flexible) arrays have 0 as their lower bound. Then, an array of size d, $0 \leq d$, is a function of type $[0, d) \rightarrow A$, where A denotes the element type of the array. As usual for functions, we call $[0, d)$ the *domain* of the array and we call values of type A *elements*. The size of an array is an attribute of that array's value, not of its type —as in Pascal—. Arrays being flexible means that different arrays may have different sizes; yet, they are all of the same type.

We denote the size of array x by $\#x$. The function $\#$ is one of the operations to be implemented. Another important operation on arrays is *element selection*, which

is the same as function application: for array x and $i, 0 \leq i < \#x$, element $x \cdot i$ is the value of x in point i of its domain. Notice that an array is completely determined by its size and its elements. In what follows we use this without explicit reference.

By means of *element replacement*, an array can be modified in a single point of its domain. For array x, natural $i, 0 \leq i < \#x$, and element b, we use $x:i,b$ to denote the array y that satisfies:

$$\#y = \#x \;\wedge\; y \cdot i = b \;\wedge\; (\forall j : 0 \leq j < \#x \wedge j \neq i : y \cdot j = x \cdot j) \quad .$$

Notice that element selection and replacement are only meaningful for nonempty arrays.

The functions le and he represent the operations to *extend* an array with an additional element at the *lower* or *higher* end of its domain; that is, for element b and array x, arrays $le \cdot b \cdot x$ and $he \cdot b \cdot x$ are the arrays y and z that satisfy:

$$\begin{array}{l} \#y = \#x{+}1 \;\wedge\; y \cdot 0 = b \;\wedge\; (\forall j : 0 \leq j < \#x : y \cdot (j{+}1) = x \cdot j) \quad , \text{ and} \\ \#z = \#x{+}1 \;\wedge\; z \cdot (\#x) = b \;\wedge\; (\forall j : 0 \leq j < \#x : z \cdot j = x \cdot j) \quad . \end{array}$$

Finally, the (partial) inverses of le and he are the functions lr and hr; they can be used to *remove* the extreme elements of an array. For array x satisfying $\#x \geq 1$, arrays $lr \cdot x$ and $hr \cdot x$ are the arrays y and z satisfying:

$$\begin{array}{l} \#y = \#x{-}1 \;\wedge\; (\forall j : 0 \leq j < \#y : y \cdot j = x \cdot (j{+}1)) \quad , \text{ and} \\ \#z = \#x{-}1 \;\wedge\; (\forall j : 0 \leq j < \#z : z \cdot j = x \cdot j) \quad . \end{array}$$

From these specifications it follows, for instance, that $lr \cdot (le \cdot b \cdot x) = x$ and that $hr \cdot (he \cdot b \cdot x) = x$.

1.1 Braun Trees

For the sake of the required (logarithmic) efficiency, we use a divide-and-conquer approach. The unique array of size 0 can be represented by the unique element of a unit type. An array x of size $d{+}1, 0 \leq d$, can be represented as follows. One element, namely $x \cdot 0$, is kept separate; the remaining elements $x \cdot (j{+}1), 0 \leq j < d$, are partitioned according to the *parity* of j. That is, we distinguish the elements $x \cdot (2{*}i{+}1)$, $0 \leq 2{*}i < d$, and the elements $x \cdot (2{*}i{+}2), 0 \leq 2{*}i{+}1 < d$. The ranges $0 \leq 2{*}i < d$ and $0 \leq 2{*}i{+}1 < d$ can be rewritten into $0 \leq i < d\mathsf{div}2{+}d\mathsf{mod}2$ and $0 \leq i < d\mathsf{div}2$ respectively. Hence, the two collections of elements thus obtained can be considered as arrays again, of sizes $d\mathsf{div}2{+}d\mathsf{mod}2$ and $d\mathsf{div}2$.

So, an array of size $d{+}1$ can be represented by a triple consisting of an element and two arrays of sizes $d\mathsf{div}2{+}d\mathsf{mod}2$ and $d\mathsf{div}2$. By applying the same trick to these two subarrays, we obtain a recursive data-type the elements of which we call *Braun trees*. We represent them by tuples, using the empty tuple $\langle\rangle$ to represent the empty array and using the triple $\langle a, s, t\rangle$ to represent the array consisting of element a and subarrays s and t.

We devote the remainder of this section to a formal definition of Braun trees and of how they represent flexible arrays. Trees are defined recursively by:

$\langle\rangle$ is a tree , and
$\langle a,s,t\rangle$ is a tree , for element a and tree s,t .

The fact that the two subtrees of $\langle a,s,t\rangle$ represent arrays of almost equal sizes gives rise to trees that are *balanced*, which can be formalised as follows. We introduce a predicate *bal* on the set of trees and we define the Braun[0] trees as those trees that satisfy *bal*. This predicate is defined in terms of the sizes of the subtrees; therefore, we denote the size of tree s by $\#s$:

$$\begin{array}{lcl} bal{\cdot}\langle\rangle & \equiv & \mathsf{true} \\ bal{\cdot}\langle a,s,t\rangle & \equiv & \#t \le \#s \ \wedge\ \#s \le \#t{+}1 \ \wedge\ bal{\cdot}s \ \wedge\ bal{\cdot}t \\ \#\langle\rangle & = & 0 \\ \#\langle a,s,t\rangle & = & 1+\#s+\#t \end{array}$$

Throughout the rest of this paper we only consider Braun trees and we use the term "tree" for "Braun tree".

We now define how trees represent arrays, by defining how the size and the elements of the array depend on the tree representing it. First, we have:

the size of the array represented by tree s equals $\#s$.

Second, we denote element i of the array represented by tree s by $s!i$; for element a and tree s,t, the elements of the array represented by $\langle a,s,t\rangle$ are defined as follows:

$$\begin{array}{lcll} \langle a,s,t\rangle\,!\,0 & = & a & \\ \langle a,s,t\rangle\,!\,(2{*}i{+}1) & = & s!i \ , & \text{for } i : 0 \le i < \#s \\ \langle a,s,t\rangle\,!\,(2{*}i{+}2) & = & t!i \ , & \text{for } i : 0 \le i < \#t \end{array}$$

Notice that the domain of the array thus represented is $[0,d{+}1)$, where $d = \#s{+}\#t$; the tripartitioning $\{0\} \cup \{i : 0 \le i < \#s : 2{*}i{+}1\} \cup \{i : 0 \le i < \#t : 2{*}i{+}2\}$ exactly characterizes this domain. That trees are balanced plays a crucial role here, as is reflected by the following property.

Property 0:

$$bal{\cdot}\langle a,s,t\rangle \ \wedge\ \#\langle a,s,t\rangle = d{+}1 \ \Rightarrow\ \#s = d\mathsf{div}2{+}d\mathsf{mod}2 \ \wedge\ \#t = d\mathsf{div}2$$

□

The size of a tree equals the size of the array represented by it; computing the size of the array in this way requires a linear amount of time, though. It is possible and sufficient to record the size of the whole array separately. As a consequence of Property 0, it is not necessary to record the size of *each* subtree together with that subtree.

As a consequence of its balance, the *height* of a tree is proportional to the logarithm of its size. This is the reason why all operations on trees require an amount of time that is at most logarithmic in the size of the tree.

[0] We use the term "Braun tree" because "balanced" is too general a notion here: Braun trees are trees that are so neatly balanced that they admit the operation of element selection, as defined in the next paragraph.

1.2 Element Selection and Replacement

In the previous section we have defined element selection. This definition is recursive and it can be considered as a (functional) program right away. We repeat it here:

$$\begin{array}{lcll} \langle a,s,t\rangle\,!\,0 & = & a & \\ \langle a,s,t\rangle\,!\,(2{*}i{+}1) & = & s!i \;\;, & \text{for } i : 0 \le i < \#s \\ \langle a,s,t\rangle\,!\,(2{*}i{+}2) & = & t!i \;\;, & \text{for } i : 0 \le i < \#t \end{array}$$

Element replacement is so similar to element selection that there is hardly anything to derive; the difference lies only in the value produced:

$$\begin{array}{lcll} \langle a,s,t\rangle : 0, b & = & \langle b,s,t\rangle & \\ \langle a,s,t\rangle : (2{*}i{+}1), b & = & \langle\, a\,,\,(s{:}i,b)\,,\,t\,\rangle \;\;, & \text{for } i : 0 \le i < \#s \\ \langle a,s,t\rangle : (2{*}i{+}2), b & = & \langle\, a\,,\,s\,,\,(t{:}i,b)\,\rangle \;\;, & \text{for } i : 0 \le i < \#t \end{array}$$

1.3 Intermezzo on Bag Insertions

Arrays as well as trees can be considered as (representations of) *bags* of elements. In terms of bags, the two array extension operations, *le* and *he*, both boil down to insertion of an element into a bag. To separate our concerns, we first investigate bag insertion in isolation.

We denote the bag represented by tree s by $[\![s]\!]$; a recursive definition for function $[\![\cdot]\!]$ is —where $\{\,\}$ denotes the *empty bag* and $+$ denotes *bag summation*—:

$$\begin{array}{lcl} [\![\langle\rangle]\!] & = & \{\,\} \\ [\![\langle a,s,t\rangle]\!] & = & \{a\} + [\![s]\!] + [\![t]\!] \end{array}$$

We now derive definitions for a function *ins*, where for element b and tree s the tree $ins{\cdot}b{\cdot}s$ is a solution of the equation[1] (with unknown u):

$$u \;:\; [\![u]\!] = \{b\} + [\![s]\!] \;\wedge\; bal{\cdot}u \;\;.$$

We use the first conjunct of this equation to guide our derivation, whereas the second conjunct remains as an a posteriori proof obligation. By induction over the size of the trees we derive, starting with the case that s is the empty tree:

$$\begin{array}{ll} & [\![u]\!] = \{b\} + [\![\langle\rangle]\!] \\ \equiv & \quad \{\ \{\,\} \text{ is the identity of } +\text{; definition of } [\![\cdot]\!]\ \} \\ & [\![u]\!] = \{b\} + [\![\langle\rangle]\!] + [\![\langle\rangle]\!] \\ \equiv & \quad \{\ \text{definition of } [\![\cdot]\!]\ \} \\ & [\![u]\!] = [\![\, \langle b, \langle\rangle, \langle\rangle\rangle \,]\!] \\ \Leftarrow & \quad \{\ \text{Leibniz}\ \} \\ & u = \langle b, \langle\rangle, \langle\rangle\rangle \;\;. \end{array}$$

[1] The word "equation" is used here for arbitrary predicates, not just equalities.

The first step of this derivation may look like a rabbit, but it is not: the only way to solve an equation of the form $u : [\![u]\!] = E$ is to transform E into an expression of the form $[\![F]\!]$ and then to apply "Leibniz", as we did in the last step. In view of the definition of $[\![\cdot]\!]$ and the occurrence of $\{b\}$, the only thing we can do is to work towards a formula of the shape $[\![\langle b, ?, ? \rangle]\!]$.

Notice that in the above derivation in all steps except the last one, only the right-hand side of the equality is manipulated. In order to avoid the continued rewriting of the constant left-hand side, we shall carry out the calculation with the right-hand expression only. That is, in order to solve an equation of the form $u : f \cdot u = E$, we transform E into an equivalent expression $f \cdot F$ in isolation, after which we conclude that F is a solution of the equation.

For the composite tree $\langle a, s, t \rangle$ we derive, in the same "goal-driven" way:

$$
\begin{array}{ll}
 & \{b\} + [\![\langle a, s, t \rangle]\!] \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & \{b\} + \{a\} + [\![s]\!] + [\![t]\!] \\
= & \quad \{ \text{ specification of } ins, \text{ by induction hypothesis (see below) } \} \\
 & \{b\} + [\![ins \cdot a \cdot s]\!] + [\![t]\!] \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & [\![\langle b , ins \cdot a \cdot s , t \rangle]\!] \ .
\end{array}
$$

The second step of this derivation represents a choice out of many possibilities; because bag summation is symmetric and associative, we have 8 possibilities here: a and b may be interchanged, s and t may be interchanged, and the recursive application of ins may be taken as the "right" or as the "left" subtree in the resulting tree. Thus, we obtain 8 different definitions for a function ins satisfying the first conjunct of the above specification.

Regarding the remaining proof obligation we observe that, by the definition of bal, $bal \cdot \langle b, \langle \rangle, \langle \rangle \rangle$ holds. For the 8 alternatives a simple calculation reveals that, generally, $bal \cdot \langle a, s, t \rangle \Rightarrow bal \cdot u$ only holds when s and t satisfy an additional precondition, as follows —notice that we need write down 4 cases only, since the relative positions of a and b are irrelevant—:

$$
\begin{array}{lcll}
bal \cdot \langle a, s, t \rangle & \Rightarrow & bal \cdot \langle b , ins \cdot a \cdot s , t \rangle & , \text{ if } \#s = \#t \\
bal \cdot \langle a, s, t \rangle & \Rightarrow & bal \cdot \langle b , ins \cdot a \cdot t , s \rangle & , \text{ if } \mathsf{true} \\
bal \cdot \langle a, s, t \rangle & \Rightarrow & bal \cdot \langle b , s , ins \cdot a \cdot t \rangle & , \text{ if } \#s = \#t + 1 \\
bal \cdot \langle a, s, t \rangle & \Rightarrow & bal \cdot \langle b , t , ins \cdot a \cdot s \rangle & , \text{ if } \mathsf{false}
\end{array}
$$

Apparently, the last alternative is never useful; thus, only 6 out of the 8 alternatives can be used when the trees are to remain balanced. As for the sizes of the trees, ins has the following property.

Property 1: for all b and s we have $\#(ins \cdot b \cdot s) = \#s + 1$

□

The definitions we shall derive for the array extension operations turn out to correspond to some of the recursive schemes discussed here. That is, these operations are refinements of the above functions *ins*. As a consequence, we have done away with the proof obligations regarding the size and the balance of the trees.

1.4 Low Extension and Removal

We recall the specification of functon *le*, but now reformulated in terms of trees. For element b and tree s, the value $le{\cdot}b{\cdot}s$ is the tree u satisfying:

$$\#u = \#s{+}1 \;\wedge\; u!0 = b \;\wedge\; (\forall j : 0 \leq j < \#s : u!(j{+}1) = s!j) \quad .$$

According to the analysis in the previous section, we have only one option for $le{\cdot}b{\cdot}\langle\,\rangle$; fortunately, it satisfies all requirements. So, we define:

$$le{\cdot}b{\cdot}\langle\,\rangle \;=\; \langle b, \langle\,\rangle, \langle\,\rangle\rangle \quad .$$

For the composite tree $\langle a, s, t\rangle$ of size $d{+}1$, the value $le{\cdot}b{\cdot}\langle a, s, t\rangle$ is the tree u satisfying:

$$\#u = d{+}2 \;\wedge\; u!0 = b \;\wedge\; (\forall j : 0 \leq j < d{+}1 : u!(j{+}1) = \langle a, s, t\rangle!j) \quad .$$

The term $\langle a, s, t\rangle!j$ and the definition of element selection suggest a 3-way case analysis in the range $0 \leq j < d{+}1$; so, using the definition of $!$, we rewrite this as:

$$\begin{array}{l} \#u = d{+}2 \;\wedge\; u!0 = b \;\wedge\; u!1 = a \;\wedge \\ (\forall i : 0 \leq i < \#s : u!(2{*}i{+}2) = s!i) \;\wedge\; (\forall i : 0 \leq i < \#t : u!(2{*}i{+}3) = t!i) \quad . \end{array}$$

This provides an explicit definition of the elements of u in terms of a, b, s, t. In view, again, of the definition of $!$, and observing that u will be a composite tree, we are forced to distinguish between $u!0$, $u!(2{*}i{+}1)$, and $u!(2{*}i{+}2)$. By comparing this with the above requirement we conclude that we must choose $u = \langle b, v, s\rangle$, where v is the tree satisfying:

$$\#v = \#t{+}1 \;\wedge\; v!0 = a \;\wedge\; (\forall i : 0 \leq i < \#t : v!(i{+}1) = t!i) \quad .$$

This specification of v is precisely the specification of the tree $le{\cdot}a{\cdot}t$; because t is smaller than $\langle a, s, t\rangle$ we may use this recursive application of *le*. Thus we obtain the following definition for *le*:

$$\begin{array}{lcl} le{\cdot}b{\cdot}\langle\,\rangle & = & \langle b, \langle\,\rangle, \langle\,\rangle\rangle \\ le{\cdot}b{\cdot}\langle a, s, t\rangle & = & \langle\, b\,,\; le{\cdot}a{\cdot}t\,,\; s\,\rangle \end{array}$$

This definition corresponds to one of the alternatives for bag insertion discussed in the previous section; as already stated there, this definition also satisfies the requirements regarding the size and the balance of the resulting tree.

Finally, we recall the specification of the function *lr*, reformulated in terms of trees. For composite tree s the value $lr{\cdot}s$ is the tree u that satisfies:

$$\#u = \#s{-}1 \;\wedge\; (\forall j : 0 \leq j < \#u : u!j = s!(j{+}1)) \quad .$$

Using that $lr{\cdot}(le{\cdot}b{\cdot}s) = s$, we obtain from the above definition for le the following definition for lr, by means of "program inversion". The verification that this definition indeed satisfies the specification requires a calculation very much like the above derivation for le.

$$\begin{array}{lcl} lr{\cdot}\langle a, \langle\,\rangle, \langle\,\rangle\rangle & = & \langle\,\rangle \\ lr{\cdot}\langle a, s, t\rangle & = & \langle\, s!0\,,\, t\,,\, lr{\cdot}s\,\rangle \quad , \text{ for } s : s \neq \langle\,\rangle \end{array}$$

1.5 High Extension and Removal

We recall the specification of function he, now reformulated in terms of trees. For element b and tree s, the value $he{\cdot}b{\cdot}s$ is the tree u satisfying:

$$\#u = \#s{+}1 \;\wedge\; u!(\#s) = b \;\wedge\; (\forall j : 0 \leq j < \#s : u!j = s!j) \quad .$$

As in the previous section, the only possible definition for $he{\cdot}b{\cdot}\langle\,\rangle$ is:

$$he{\cdot}b{\cdot}\langle\,\rangle = \langle b, \langle\,\rangle, \langle\,\rangle\rangle \quad .$$

For the composite tree $\langle a, s, t\rangle$ we observe —calculation omitted— that the first and the third conjuncts of the above specification can be met both by:

$$\begin{array}{l} he{\cdot}b{\cdot}\langle a, s, t\rangle = \langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle \quad , \text{ and by:} \\ he{\cdot}b{\cdot}\langle a, s, t\rangle = \langle\, a\,,\, s\,,\, he{\cdot}b{\cdot}t\,\rangle \quad . \end{array}$$

To investigate which one we need, we try to prove the second conjunct of the specification, where we assume $\#\langle a, s, t\rangle = d{+}1$; recall that then $\#s = d\mathsf{div}2{+}d\mathsf{mod}2$ and $\#t = d\mathsf{div}2$. For the first alternative to satisfy the second conjunct, we need:

$$\begin{array}{ll} & \langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle\,!\,(d{+}1) \\ = & \quad \{\ \text{assume } d \text{ to be even, set } d{=}2{*}e\text{; definition of } !\ \} \\ & he{\cdot}b{\cdot}s\,!\,e \\ = & \quad \{\ d{=}2{*}e\text{, so } \#s{=}e\text{; specification of } he\text{, by ind.hyp.}\ \} \\ & b \quad . \end{array}$$

So, the expression $\langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle$ satisfies the specification if d is even, provided that we also have $bal{\cdot}\langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle$; the condition for this is $\#s{=}\#t$, which is equivalent to d being even.

Similarly, we can derive that the other expression, $\langle\, a\,,\, s\,,\, he{\cdot}b{\cdot}t\,\rangle$, satisfies the specification if d is odd; in that case we have $\#s{=}\#t{+}1$, which is exactly the condition for $bal{\cdot}\langle\, a\,,\, s\,,\, he{\cdot}b{\cdot}t\,\rangle$.

Thus, we obtain the following definition for he:

$$\begin{array}{lcl} he{\cdot}b{\cdot}\langle\,\rangle & = & \langle b, \langle\,\rangle, \langle\,\rangle\rangle \\ he{\cdot}b{\cdot}\langle a, s, t\rangle & = & \mathsf{if}\ d\mathsf{mod}2 = 0 \;\rightarrow\; \langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle \\ & & \;[]\ d\mathsf{mod}2 \neq 0 \;\rightarrow\; \langle\, a\,,\, s\,,\, he{\cdot}b{\cdot}t\,\rangle \\ & & \mathsf{fi}\ \mathsf{where}\ d = \#\langle a, s, t\rangle - 1\ \mathsf{end} \end{array}$$

To allow for logarithmic computation times, the value $\#\langle a,s,t\rangle$ occurring in this definition must not be computed but must be supplied as an additional parameter instead. By Property 0 this is possible.

Finally, we recall the specification of the function hr, reformulated in terms of trees. For composite tree s the value $hr{\cdot}s$ is the tree u that satisfies:

$$\#u = \#s-1 \;\wedge\; (\forall j : 0\le j<\#u : u!j = s!j) \quad .$$

Using that $hr{\cdot}(he{\cdot}b{\cdot}s) = s$, we can derive the following definition for hr from the above one for he, again by program inversion:

$$\begin{array}{lcl} hr{\cdot}\langle a,\langle\rangle,\langle\rangle\rangle & = & \langle\rangle \\ hr{\cdot}\langle a,s,t\rangle & = & \textsf{if } d\textsf{mod}2 = 0 \;\rightarrow\; \langle\, a\,,\, hr{\cdot}s\,,\, t\,\rangle \\ & & \;[]\; d\textsf{mod}2 \neq 0 \;\rightarrow\; \langle\, a\,,\, s\,,\, hr{\cdot}t\,\rangle \\ & & \textsf{fi where } d = \#\langle a,s,t\rangle - 2 \textsf{ end} \quad , \text{ for } s : s\neq\langle\rangle \end{array}$$

1.6 Summary of the Functional Programs

$$\begin{array}{lcl} \langle a,s,t\rangle\,!\,0 & = & a \\ \langle a,s,t\rangle\,!\,(2{*}i{+}1) & = & s!i \quad , \quad \text{for } i : 0\le i<\#s \\ \langle a,s,t\rangle\,!\,(2{*}i{+}2) & = & t!i \quad , \quad \text{for } i : 0\le i<\#t \end{array}$$

$$\begin{array}{lcl} \langle a,s,t\rangle : 0,b & = & \langle b,s,t\rangle \\ \langle a,s,t\rangle : (2{*}i{+}1),b & = & \langle\, a\,,\,(s:i,b)\,,\,t\,\rangle \quad , \quad \text{for } i : 0\le i<\#s \\ \langle a,s,t\rangle : (2{*}i{+}2),b & = & \langle\, a\,,\,s\,,\,(t:i,b)\,\rangle \quad , \quad \text{for } i : 0\le i<\#t \end{array}$$

$$\begin{array}{lcl} le{\cdot}b{\cdot}\langle\rangle & = & \langle b,\langle\rangle,\langle\rangle\rangle \\ le{\cdot}b{\cdot}\langle a,s,t\rangle & = & \langle\, b\,,\, le{\cdot}a{\cdot}t\,,\, s\,\rangle \end{array}$$

$$\begin{array}{lcl} lr{\cdot}\langle a,\langle\rangle,\langle\rangle\rangle & = & \langle\rangle \\ lr{\cdot}\langle a,s,t\rangle & = & \langle\, s!0\,,\, t\,,\, lr{\cdot}s\,\rangle \quad , \text{ for } s : s\neq\langle\rangle \end{array}$$

$$\begin{array}{lcl} he{\cdot}b{\cdot}\langle\rangle & = & \langle b,\langle\rangle,\langle\rangle\rangle \\ he{\cdot}b{\cdot}\langle a,s,t\rangle & = & \textsf{if } d\textsf{mod}2 = 0 \;\rightarrow\; \langle\, a\,,\, he{\cdot}b{\cdot}s\,,\, t\,\rangle \\ & & \;[]\; d\textsf{mod}2 \neq 0 \;\rightarrow\; \langle\, a\,,\, s\,,\, he{\cdot}b{\cdot}t\,\rangle \\ & & \textsf{fi where } d = \#\langle a,s,t\rangle - 1 \textsf{ end} \end{array}$$

$$\begin{array}{lcl} hr{\cdot}\langle a,\langle\rangle,\langle\rangle\rangle & = & \langle\rangle \\ hr{\cdot}\langle a,s,t\rangle & = & \textsf{if } d\textsf{mod}2 = 0 \;\rightarrow\; \langle\, a\,,\, hr{\cdot}s\,,\, t\,\rangle \\ & & \;[]\; d\textsf{mod}2 \neq 0 \;\rightarrow\; \langle\, a\,,\, s\,,\, hr{\cdot}t\,\rangle \\ & & \textsf{fi where } d = \#\langle a,s,t\rangle - 2 \textsf{ end} \quad , \text{ for } s : s\neq\langle\rangle \end{array}$$

2 Sequential Implementations

In this section we derive sequential implementations from the recursive function definitions in the previous section. We represent trees by data structures built from smaller units called *nodes*, which are linked together by means of *pointers*. For this purpose we use a program notation that is a mixture of guarded commands and Pascal.

Before doing so, however, we explain the techniques used for this transformation by means of a simple example. Readers who are sufficiently familiar with techniques for pointer manipulation and recursion elimination may wish to skip the next subsection at first reading.

2.0 A Few Simple Transformation Techniques

This subsection consists of two parts. First, we present two (well-known) instances of how tail-recursive definitions can be transformed into equivalent non-recursive programs. Second, we illustrate how a simple function on lists, namely list catenation, can be implemented as a sequential program.

* * *

We consider the following tail-recursive definition of a function F:

$$
\begin{array}{lll}
F{\cdot}x & = & \mathsf{if}\ \neg b{\cdot}x \ \rightarrow\ f{\cdot}x \\
 & & []\ \ b{\cdot}x \ \rightarrow\ F{\cdot}(g{\cdot}x) \\
 & & \mathsf{fi}
\end{array}
$$

From this definition we obtain the following iterative, sequential program for the computation of, say, $F{\cdot}X$. The correctness of this program follows from the invariance of $F{\cdot}X = F{\cdot}x$:

$$
\begin{array}{l}
\ \ x := X \\
;\ \mathsf{do}\ b{\cdot}x \ \rightarrow\ x := g{\cdot}x\ \ \mathsf{od} \\
;\ r := f{\cdot}x \\
\{\ r = F{\cdot}X\ \}
\end{array}
$$

Next, we consider the following tail-recursive procedure P, in which x is assumed to be a value parameter:

$$
\begin{array}{l}
\mathsf{procedure}\ P(x) \\
=\ [\![\ \mathsf{if}\ \neg b{\cdot}x \ \rightarrow\ S_0 \\
\qquad [] \ \ b{\cdot}x \ \rightarrow\ S_1\ ;\ P(g{\cdot}x) \\
\qquad \mathsf{fi} \\
\ \ ]\!]
\end{array}
$$

This procedure can be transformed into the following equivalent non-recursive one:

$$
\begin{array}{l}
\mathsf{procedure}\ P(x) \\
=\ [\![\ \mathsf{do}\ b{\cdot}x \ \rightarrow\ S_1\ ;\ x := g{\cdot}x\ \ \mathsf{od} \\
\quad ;\ S_0 \\
\ \ ]\!]
\end{array}
$$

If we wish to get rid of the procedure altogether, each call $P(X)$ may be replaced by the following program fragment; we call this *unfolding* $P(X)$:

```
|[ var x ;
   x := X
 ; do b·x → S1 ; x := g·x od
 ; S0
]|
```

* * *

We consider the function f that maps two lists onto their catenation, defined recursively as follows:

```
f·x·y = if x = [] → y
        [] x ≠ [] → hd·x cons f·(tl·x)·y
        fi
```

The first step towards a sequential implementation is to recode this definition as a procedure definition; the reason for using a result parameter for the function result will become clear later:

```
procedure P0(x, y; result z)
= |[ { post: z = f·x·y }
     var h ;
     if x = [] → z := y
     [] x ≠ [] → P0(tl·x , y , h)
                 ; z := hd·x cons h
     fi
  ]|
```

In the second step we introduce the representation of lists by means of *nodes* and *pointers*. A list is represented by a pointer; a pointer is either nil , or a reference to a node —Pascal: a record— consisting of an element and a pointer. The value nil has as its only property that it differs from all pointer values referring to nodes. By means of a call of the standard procedure $new(p)$, pointer variable p is assigned a value that differs from nil and that differs from all pointer values "currently in use": we call such a pointer value, and the node referred to by it, *fresh*.

The type definitions needed to define this data structure formally are:

```
type listp = pointer to node ;
     node = ⟨ hd : element ; tl : listp ⟩
```

For p a pointer of type *listp* and $p \neq$ nil , we use $p\uparrow$ to denote the node to which p refers. We denote the two components of this node by $p\uparrow\cdot hd$ and $p\uparrow\cdot tl$; that is, we use the Pascal convention of *field selectors* to identify the components of a pair. As in Pascal, we admit and shall use assignments to individual components of nodes.

A pointer p represents a list $[\![p]\!]$, say, as follows:

```
[[ nil ]] = []
[[ p ]]   = p↑·hd cons [[ p↑·tl ]] , for p : p ≠ nil
```

(The use of nil to represent the (one and only) empty list is somewhat opportunistic, but it simplifies things a little.)

By incorporation of this list representation we obtain from the above procedure P_0 our next version; notice that $[\![p]\!]=[] \equiv p=\mathsf{nil}$:

```
procedure P1(p, q; result r)
= |[ { post: [[r]] = f·[[p]]·[[q]] }
     var h;
     if p=nil  → r := q
     [] p≠nil  → P1(p↑·tl, q, h)
                 ; new(r)
                 ; r↑ := ⟨p↑·hd, h⟩
     fi
  ]|
```

Procedure P_1 entails an important design decision. In this procedure the only assignment to values of type *node* is the assigment to $r\uparrow$, which is a fresh node. As a result, existing nodes are not modified. Generally, *if the value of a node*, once it has been created and initialised, *is never changed, then this node may be freely shared* among different data structures. If such a node contains pointers to other nodes, then the "never change" condition must also hold for all nodes that are reachable from this node. By building data structures in this way, the use of sharing saves both storage space and computation time. For example, a *list assignment* can now be implemented by copying a pointer only, which is an $\mathcal{O}(1)$ operation, instead of by copying the whole list. As a result, the two lists are represented in storage by the same data structure as long as they remain equal. In procedure P_1 , for instance, we have used the pointer assignment $r := q$ to implement the list assignment $z := y$ from procedure P_0 . The price to be paid for this flexibility is that efficient storage management involves some form of garbage collection.

Aside: As a matter of fact, sharing is possible when, for every pointer p representing a list, the value $[\![p]\!]$ is never changed. This requirement is weaker than the requirement that $p\uparrow$ is never changed. That is, nodes may be *overwritten* as long they still represent the same abstract values —in our case: lists— . For example, node overwriting is frequently used in graph-reduction machines. In this paper, we can live with the stricter regime.

□

In the third step we employ the possibility to use assignments to individual components of nodes, to rearrange the order of the assignments in such a way that the procedure becomes tail-recursive. Thus, we obtain:

```
procedure P2(p, q; result r)
= |[ { post: [[r]] = f·[[p]]·[[q]] }
     if p=nil  → r := q
     [] p≠nil  → new(r)
                 ; r↑·hd := p↑·hd
                 ; P2(p↑·tl, q, r↑·tl)
     fi
  ]|
```

Procedure P_2 is tail-recursive but it still contains a result parameter; the transformation into iterative form discussed in the beginning of this subsection is only applicable to procedures with value parameters only. Result parameter r may be turned into a value parameter, provided that we remove all assignments to r from the procedure. (Notice that assignments to $r\uparrow$ are not assignments to r.) The assignment $r := q$ can be removed by the introduction of an additional procedure P_3 that is identical to P_2 but with its precondition strengthened with $p \neq \mathsf{nil}$. This requires, of course, that the case $p = \mathsf{nil}$ be dealt with separately. Similarly, the assignment $new(r)$ can be eliminated by strengthening the precondition of P_3 even further, namely with "r is fresh". Thus, we obtain:

```
procedure P2(p, q; result r)
= |[ { post: [[r]] = f·[[p]]·[[q]] }
     if p = nil  → r := q
     [] p ≠ nil  → new(r)
                   { p ≠ nil ∧ "r is fresh" }
                   ; P3(p, q, r)
     fi
   ]|

procedure P3(p, q, r)
= |[ { pre: p ≠ nil ∧ "r is fresh" }
     { post: [[r]] = f·[[p]]·[[q]] }
     r↑·hd := p↑·hd
   ; if p↑·tl = nil  → r↑·tl := q
     [] p↑·tl ≠ nil  → new(r↑·tl)
                     ; P3(p↑·tl, q, r↑·tl)
     fi
   ]|
```

Finally, if we now distribute the assignment $r\uparrow \cdot hd := p\uparrow \cdot hd$ over the alternatives of the succeeding selection statement, procedure P_3 can be transformed into iterative form; by unfolding its one and only call, P_3 can be eliminated altogether. This yields our final, iterative implementation of list catenation:

```
procedure P4(p, q; result r)
= |[ { post: [[r]] = f·[[p]]·[[q]] }
     var h;
     if p = nil  → r := q
     [] p ≠ nil  → new(r) ; h := r
                   { invariant: p ≠ nil ∧ "h is fresh" }
                   ; do p↑·tl ≠ nil → h↑·hd := p↑·hd
                                    ; new(h↑·tl)
                                    ; p, h := p↑·tl, h↑·tl
                     od
                   ; h↑ := ⟨p↑·hd, q⟩
     fi
   ]|
```

2.1 Selection and Replacement

We represent trees by means of data structures composed from nodes and pointers. A tree is represented by a pointer; a pointer is either nil or a pointer to a node consisting of an element and 2 pointers. The value nil has as its only property that it differs from all pointer values referring to nodes. By means of a call of the standard procedure *new*(p), pointer variable p is assigned a value that differs from nil and that differs from all pointer values "currently in use": we call such a pointer value, and the node referred to by it, *fresh*.

Assignments to nodes will be restricted to fresh nodes; as a result, the values of existing nodes will never be changed and we may employ *node sharing*. (See the previous subsection, for a slightly more elaborate discussion of node sharing.)

The type definitions needed to define the data structure formally are:

```
type  treep = pointer to node ;
      node  = ⟨ a : element ; s,t : listp ⟩
```

For p a pointer of type *treep* and $p \neq$ nil, we use $p\uparrow$ to denote the node to which p refers. We denote the three components of this node by $p\uparrow{\cdot}a$, $p\uparrow{\cdot}s$, and $p\uparrow{\cdot}t$. We admit and shall use assignments to individual components of nodes.

A pointer p represents a tree $[\![p]\!]$, say, as follows:

$$[\![\,\mathsf{nil}\,]\!] = \langle\rangle$$
$$[\![p]\!] = \langle p\uparrow{\cdot}a\,,\,[\![p\uparrow{\cdot}s]\!]\,,\,[\![p\uparrow{\cdot}t]\!]\rangle \quad \text{, for } p : p \neq \mathsf{nil}$$

The definition of element selection is tail-recursive. Such a definition can be transformed into an iterative program in a straightforward way. Next, the tree operations in terms of tuples are recoded in terms of node and pointer operations. Thus, we obtain the following program; to make verification of this transformation somewhat easier, we repeat the definition of element selection here with some of the syntactic sugar —the parameter patterns— removed:

```
u!i = if i=0 → u·a
      [] i>0 → if jmod2=0 → u·s!(jdiv2)
                [] jmod2≠0 → u·t!(jdiv2)
                fi where j=i−1 end
      fi

procedure EL(p, i; result e)
= |[ { pre: 0≤i<#[[p]] }
     { post: e = [[p]]!i (initial values of p, i) }
     do i≠0 → i:=i−1
            ; if imod2=0 → p, i := p↑·s , idiv2
              [] imod2=1 → p, i := p↑·t , idiv2
              fi
     od
   ; e := p↑·a
  ]|
```

For the implementation of element replacement, we introduce a procedure *REP* with the following specification:

```
procedure REP(p, i, b; result r)
= |[ { pre:  0≤i<#[[p]] }
     { post: [[r]] = [[p]] : i, b }
  ]|
```

As a first approximation, we construct the following (recursive) code from the (recursive) definiton of element replacement:

```
procedure REP(p, i, b; result r)
= |[ var h;
     if i=0 → new(r) ; r↑ := ⟨b, p↑·s, p↑·t⟩
     [] i≠0 → i := i−1
               ; if imod2=0 → REP(p↑·s, idiv2, b, h)
                              ; new(r) ; r↑ := ⟨p↑·a, h, p↑·t⟩
                 [] imod2=1 → REP(p↑·t, idiv2, b, h)
                              ; new(r) ; r↑ := ⟨p↑·a, p↑·s, h⟩
                 fi
     fi
  ]|
```

By means of the techniques from the previous section, this procedure can be transformed into the following iterative one:

```
procedure REP1(p, i, b; result r)
= |[ var h;
     new(r) ; h := r
   ; do i≠0 → i := i−1
               ; if imod2=0 → h↑·a := p↑·a ; h↑·t := p↑·t
                              ; new(h↑·s) ; p, i, h := p↑·s, idiv2, h↑·s
                 [] imod2=1 → h↑·a := p↑·a ; h↑·s := p↑·s
                              ; new(h↑·t) ; p, i, h := p↑·t, idiv2, h↑·t
                 fi
     od
   ; h↑ := ⟨b, p↑·s, p↑·t⟩
  ]|
```

Procedure REP_1 constructs a new tree that shares the majority of its nodes with the old tree. As a result, this operation requires at most logarithmic computation time.

2.2 Extension and Removal

The techniques used in the previous subsections can be used to derive iterative implementations of the extension and removal operations as well. Although these cases are slightly more complicated, this transformation offers no further surprises;

therefore, we only present the resulting programs. As is the case with element replacement, the following procedures construct, in logarithmic time, new trees that share the majority of their nodes with the old ones.

```
procedure LE(p, b; result r)
= |[ var h;
     new(r) ; h := r
   ; do p ≠ nil → h↑·a := b ; h↑·t := p↑·s
                ; new(h↑·s) ; p, b, h := p↑·t, p↑·a, h↑·s
     od
   ; h↑ := ⟨b, nil, nil⟩
   ]|

procedure HE(p, d, b; result r)
= |[ { pre: d = #[[p]] }
     var h;
     new(r) ; h := r
   ; do d ≠ 0 → d := d−1
              ; if dmod2 = 0 → h↑·a := p↑·a ; h↑·t := p↑·t
                             ; new(h↑·s) ; p, d, h := p↑·s, ddiv2, h↑·s
                [] dmod2 = 1 → h↑·a := p↑·a ; h↑·s := p↑·s
                             ; new(h↑·t) ; p, d, h := p↑·t, ddiv2, h↑·t
                fi
     od
   ; h↑ := ⟨b, nil, nil⟩
   ]|

procedure HR(p, d; result r)
= |[ { pre: d = #[[p]] }
     var h;
     if d = 1 → r := nil
     [] d > 1 → new(r) ; d, h := d−2, r
              ; do d ≥ 2 ∧ dmod2 = 0 → h↑·a := p↑·a ; h↑·t := p↑·t
                                     ; new(h↑·s)
                                     ; p, d, h := p↑·s, ddiv2 − 1, h↑·s
                [] d ≥ 2 ∧ dmod2 = 1 → h↑·a := p↑·a ; h↑·s := p↑·s
                                     ; new(h↑·t)
                                     ; p, d, h := p↑·t, ddiv2 − 1, h↑·t
                od
              ; if d = 0 → h↑ := ⟨p↑·a, nil, nil⟩
                [] d = 1 → h↑ := ⟨p↑·a, p↑·s, nil⟩
                fi
     fi
   ]|
```

```
procedure LR(p; result r)
= |[ var h;
     if p↑·s = nil  →  r := nil
     [] p↑·s ≠ nil  →  new(r) ; h := r
                     ; do p↑·s↑·s ≠ nil  →  h↑·a := p↑·s↑·a ; h↑·s := p↑·t
                                          ; new(h↑·t) ; p, h := p↑·s , h↑·t
                       od
                     ; h↑ := ⟨ p↑·s↑·a , p↑·t , nil ⟩
     fi
  ]|
```

3 Concluding Remarks

The introduction of Braun trees to represent flexible arrays constitutes a major design decision. Once this decision has been taken, functional programs for the array operations can be derived smoothly. These functional programs are compact and provide a good starting point for the construction of a sequential, nonrecursive implementation of the array operations. Although the resulting programs are nontrivial they can be obtained by systematic transformation of the functional programs; this requires great care but little ingenuity.

The exercise in this paper also shows that recursive data types, such as Braun trees, can be implemented by means of nodes and pointers in a simple and systematic way. Notice that we have not employed an (elaborate) formal theory on the semantics of pointer operations; nevertheless, we are convinced of the correctness of the programs thus obtained. The programs derived in this paper employ sharing of nodes by adhering to the "never-change-an-existing-node" discipline; it is equally well possible to derive programs that, instead of building a new tree, modify the existing tree by modifying nodes.

Braun trees admit several variations and embellishments. To mention a few: instead of binary trees, k-ary trees, for some fixed k, can be used. Operations like mod2 and div2 then become mod$k$ and div$k$. Larger values of k give rise to trees of smaller height —when the size remains the same—. The price to be paid is that storage utilisation decreases: each node now contains k pointers per array element instead of 2. To compensate for this, it is possible to store several consecutive array elements, instead of a single one, per node of the tree. This reduces the height of the tree even further and improves storage utilisation (, except for very small arrays). Note, however, that in a setting where nodes are shared, large nodes are awkward, because of the time needed to make copies of nodes; this places an upper bound on what can be considered as a reasonable node size.

acknowledgement

To Anne Kaldewaij and the members of the Kleine Club, for their constructive comments on an earlier version of this paper.

References

[0] Braun, W., Rem, M.: A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology (1983).
[1] Dijkstra, Edsger W.: A discipline of programming. Prentice-Hall, Englewood Cliffs (1976).

1992.4.6
for MPC Oxford, summer 1992.

This article was processed using the LaTeX macro package with LLNCS style